CAFE: An Industrial-Strength Algebraic Formal Method

Editors
K. Futatsugi
A.T. Nakagawa
T. Tamai

Elsevier

CAFE: An Industrial-Strength Algebraic Formal Method

This Page Intentionally Left Blank

CAFE: An Industrial-Strength Algebraic Formal Method

Edited by

K. Futatsugi
Japan Advanced Institute for Science and Technology, Tatsunokuchi, Japan

A.T. Nakagawa
Software Research Associates Inc., Tokyo, Japan

T. Tamai
University of Tokyo, Tokyo, Japan

2000

ELSEVIER

Amsterdam - Lausanne - New York - Oxford - Shannon - Singapore - Tokyo

ELSEVIER SCIENCE B.V.
Sara Burgerhartstraat 25
P.O. Box 211, 1000 AE Amsterdam, The Netherlands

First edition 2000

Library of Congress Cataloging-in-Publication Data
A catalog record from the Library of Congress has been applied for.

ISBN: 0-444-50556-3

♾ The paper used in this publication meets the requirements of ANSI/NISO Z39.48-1992 (Permanence of Paper).

Printed in The Netherlands

Preface

For about two years from the summer of 1996, a project called "Cafe" was undertaken by researchers from more than 10 affiliations. The major aim of the project was to launch an algebraic specification language CafeOBJ onto the industrial scene. This aim motivated several diverse activities, both theoretical and technical. On the theoretical side, novel semantic paradigms for behavioural specifications were investigated, and ways of easing proof constructions, using internet and Web technologies, were established. On the technical side, integrated systems for specification development, with emphasis both on ease of use and on proof assistance, were implemented. In addition, several case studies of significant size were pursued.

To celebrate the successful conclusion of the project, a symposium was held in Numazu, a seaside town famous for fresh seafoods, in April, 1998. On this occasion, not only the participants of the project, but many researchers in diverse fields including algebraic specifications, term rewriting, theorem proving, rewriting logic, and category theory, got together to have intensive yet relaxed presentations and discussions that lasted four days.

This book contains the selected papers by the participants of the symposium. These papers deal with various logics and formalisms underlying CafeOBJ, and/or with software development environments suitable for embedding it. Relevant keywords here are: (basic) algebraic specification, behavioural specification/logic, rewriting specification/logic, order-sorted specification/logic, executable specification, distributed computing, theorem prover interfaces.

The paper by Manuel Clavel et al. presents a rewriting logic at work. An important feature of rewriting logic is the existence of a finitely presented universal theory, where an object-level theory can be encoded and reasoned about as a plain object. Such an encoding itself can be encoded and reasoned about at a yet higher level. Thus an arbitrarily higher "reflexive tower" may be built for describing and reasoning about a vast range of problems. The paper shows two substantial examples — an inductive theorem prover and a Church-Rosser checker – that exploit this power of reflection. Moreover, these examples are written entirely within Maude, a language based on rewriting logic with built-in meta-level reasoning facilities, and are realised as executable tools.

The paper by Alexander Knapp shows another potential of rewriting logic. He has encoded a substantial fragment of Unified Modelling Language (UML) into rewriting logic. The features relevant for this case study are that rewriting logic is a natural forum where object-oriented programming and nondeterministic, concurrent computation are presented, reasoned about, and executed. In this example, the static aspects of UML collaborations are formulated as relations between objects and their dynamic aspects are captured by concurrent rewrite rules. Like the work reported in the paper by Manuel Clavel et al., this work also resulted into an executable specification (written in CafeOBJ in this case). The paper shows, in addition, that the specification is correct with respect to a temporal logic formalisation of UML.

The paper by Razvan Diaconescu et al. also highlights usages of rewriting logic, as well as behavioural logic, underlying CafeOBJ. The paper first explains how rewriting logic can

be used to state and reason about algorithms. Next it shows how both rewriting logic and behavioural logic can be used to describe nondeterminism, and argues that for proving typical properties the latter is a better alternative. It then shows further advantages of behavioural logic, by showing that the proofs formulated in the logic is much simpler than those formulated in traditional data-type specifications. It concludes by showing how to build a complex system from basic ones, in the framework of behavioural logic.

The paper by Hirotaka Ohkubo et al., in turn, exploits the order-sorted logic behind CafeOBJ to formalise a semantics of an object-oriented programming language in the paradigm of algebraic specification. The paper gives an algorithm to construct an algebraic specification from an object-oriented programme in such a way that the original programme is a correct implementation of the specification. During such a construction, order-sortedness plays a crucial rôle in dealing with polymorphism and inheritance: since an expression may evaluate to different types, it is necessary to consider and express a union type. Using order-sorted logic of CafeOBJ, the paper overcomes this problem.

The order-sorted fragment of CaOBJ is also the main concern of the paper by Peter Mosses. It compares in detail two specification languages CASL and CafeOBJ, explains where and how differences came from, and suggests possibilities of enhancing the two languages by incorporating missing features from each other. Some major differences are: absence of partial operators in CafeOBJ; predicates as Boolean operators versus predicates as such; sharing or non-sharing of parametric symbols; absence of labelled parameters in CASL; admissible formulae; lack of mechanism for restricting exportable symbols in CafeOBJ. In spite of these differences, the paper finds that the two languages share a large part in common and can gain benefits from each other, especially in incorporating convenient shorthand notations.

The paper by Masaki Ishiguro et al. presents a proof assistance system for the equational fragment of CafeOBJ. It first considers semantic constraints imposed by a couple of CafeOBJ declarations, such as views, and then tries to formulate those constraints within the syntax of CafeOBJ. It then reports a tool implementation that, under some restrictions, extract those constraints in CafeOBJ and generate proof scores thereof. It also considers a way to state a theorem of a CafeOBJ module as a semantic constraint of a CafeOBJ declaration, which makes it possible to use the tool for a more general purpose. The paper illustrates the ideas and the tool by an example involving a parameterised module.

The paper by Akishi Seo et al. gives a summary of how an integrated specification development environment was constructed, based on a paradigm called proof-as-editing. In this paradigm, specifications, theorems, proofs, and various annotations are put in documents under a uniform format, so that specifications are developed using documents and tools scattered over a network. The paper put the paradigm into a concreate form, by first desigining an extension to HTML, and then building tools that manipulate files written in the format. A major feature of this implementation is that it allows access through firewalls, so that commercial sites can exploit the technology easily.

The paper by Joseph Goguen et al. also gives a summary of such an integrated environment, but using quite different concepts and putting the emphasis on collaborative aspects of proof construction. The paper comes out of a broad-spectrum project that includes developing behavioural logic based on hidden algebra and proof methodologies

based on coinduction. The paper itself concentrates on the aspects of tool construction. In particular, it explains how a proof assistant system was designed and implemented with meticulous attention to the ease of the user interface. Some major features are: a novel graph structure used in the proof database; automatic generation of documentations in XML and HTML; and semiotic and narratological considerations.

The paper by Tohru Ogawa et al. shows a different aspect of specification development environments for CafeOBJ. It concentrates on the possibilities of visualisation. Under a system called CafePie, a CafeOBJ module is presented as a collection of iconic notations, and is edited by standard drag-and-drop operations. By default, terms are represented as trees, as usual. Based on those notations, it visualises a term rewriting process by showing its trace both as an animation and as a one-picture summary. The system also allows you to customise the representation of terms: in the example of the paper, a stack is first visualised as a stack of boxes in the literal sense, and then as queuing people waiting for a blocked exit door.

We would like to acknowledge helping hands from the referees listed below (apart from us), whose precious time was spared for reading various drafts of the submissions. Without their comments for improvement, the qualities of some of the papers included in this volume would have remained questionable. We also like to thank people at Elsevier, including Ivar Brinkman and Mara Vos-Sarmiento, who were patiently waiting for the end of our slow, winding publication schedule.

List of Referees

Răzvan Diaconescu (IMAR, Romania)
Kokichi Futatsugi (JAIST, Japan)
Joseph Goguen (UCSD, USA)
Jean-Pierre Jouannaud (Université Paris-Sud, France)
José Meseguer (SRI International, USA)
Ataru Nakagawa (SRA, Japan)
Tetsuo Tamai (University of Tokyo, Japan)
Martin Wirsing (LMUM, Germany)

Spring 2000

Kokichi Futatsugi
Ataru T. Nakagawa
Tetsuo Tamai

This Page Intentionally Left Blank

List of Authors

Manuel Clavel	Departimento de Filosofía. Universidad de Navarra. Pamplona. Spain
Răzvan Diaconescu	Institute of Mathematics of the Romanian Academy. Bucharest. Romania
Francisco Durán	Departimento de Lenguajes y Ciencias de la Computación. E.T.S.I. Informatica. Málaga. Spain
Steven Eker	Computer Science Laboratory. SRI International. Menlo Park. USA
Kokichi Futatsugi	Japan Advanced Institute for Science and Technology. Tatsunokuchi. Japan
Joseph Goguen	Department of Computer Science and Engineering. University of California. San Diego. USA
Shusaku Iida	Japan Advanced Institute for Science and Technology. Tatsunokuchi. Japan
Yasuyoshi Inagaki	Graduate School of Engineering, Nagoya University, Nagoya, Japan
Masaki Ishiguro	Information Technologies Development Department. Mitsubishi Research Institute, Tokyo, Japan
Alexander Knapp	Ludwig-Maximilians-Universität München, Munich, Germany
Kai Lin	Department of Computer Science and Engineering, University of California, San Diego, USA
José Meseguer	Computer Science Laboratory, SRI International, Menlo Park, USA
Akira Mori	Japan Advanced Institute for Science and Technology, Tatsunokuchi, Japan
Peter D. Mosses	BRICS & Department of Computer Science, University of Aarhus, Aarhus, Denmark
Ataru T. Nakagawa	Software Engineering Laboratry, SRA, Tokyo, Japan
Tohru Ogawa	Institute of Information Sciences and Electronics, University of Tsukuba, Tsukuba, Tsukuba, Japan
Hirotaka Ohkubo	Faculty of Information Science and Technology, Aichi Prefectural University, Nagoya, Japan
Grigore Roşu	Department of Computer Science and Engineering, University of California, San Diego, USA
Toshiki Sakabe	Graduate School of Engineering, Nagoya University, Nagoya, Japan
Akishi Seo	Middleware System Department, Nihon Unisys, Tokyo, Japan
Jiro Tanaka	Institute of Information Sciences and Electronics, University of Tsukuba, Tsukuba, Tsukuba, Japan
Bogdan Warinschi	Department of Computer Science and Engineering, University of California, San Diego, USA

This Page Intentionally Left Blank

Contents

Chapter 1

Building Equational Proving Tools by Reflection in Rewriting Logic*

M. Clavel[a] and F. Durán[b] and S. Eker[c] and J. Meseguer[c]

[a]Univ. de Navarra, Spain

[b]Univ. de Málaga, Spain

[c]SRI International, USA

This paper explains the design and use of two equational proving tools, namely an inductive theorem prover—to prove theorems about equational specifications with an initial algebra semantics—and a Church-Rosser checker—to check whether such specifications satisfy the Church-Rosser property. These tools can be used to prove properties of order-sorted equational specifications in CafeOBJ [17] and of membership equational logic specifications in Maude [9,12]. The tools have been written entirely in Maude and are in fact *executable specifications* in rewriting logic of the formal inference systems that they implement. The fact that rewriting logic is a reflective logic, and that Maude efficiently supports reflective rewriting logic computations is systematically exploited in the design of the tools.

1. Introduction

This paper explains the design and use of two proving tools, namely an inductive theorem prover—to prove theorems about equational specifications with an initial algebra semantics—and a Church-Rosser checker—to check whether such specifications satisfy the Church-Rosser property. These tools have been developed as part of the Cafe project [6, 20], and have been integrated, thanks to the efforts of our colleagues in Japan, within the overall Cafe environment. Through that integration, they can be used to prove formal properties of order-sorted equational specifications in CafeOBJ [17]. They can also be used on their own to prove properties of equational specifications in Maude [9]; that is, of its so-called functional modules, that are equational specifications in membership equational logic. An important feature of these tools is that they are written entirely in Maude and are in fact *executable specifications* in rewriting logic of the formal inference systems that they implement.

Both tools treat equational specifications as *data* that is manipulated. For example, the inductive theorem prover may add an induction hypothesis as a new equational axiom;

*Supported by the Advanced Software Enrichment Project of the Information-Technology Promotion Agency, Japan (IPA), DARPA and NASA through Contract NAS2-98073, and by Office of Naval Research Contracts N00014-95-C-0225 and N00014-96-C-0114.

and the Church-Rosser checker computes critical pairs and membership assertions by inspecting the equations in the original specification. This makes a *reflective* design—in which theories become data at the metalevel—ideally suited for the task. Indeed, the fact that rewriting logic is a reflective logic [7,14], and that Maude efficiently supports reflective rewriting logic computations is systematically exploited in the tools [8,11].

Rewriting logic reflection has one important additional advantage, namely, the modular and declarative treatment of *proof tactics* or *strategies* [14,7]. Such tactics can be formally specified by rewrite rules at the metalevel of the theory expressing the tool's inference system, and can be easily changed at will without any modification whatsoever to the inference system itself.

The very high level of abstraction at which the tools have been developed has made it relatively easy for us to build them, makes understanding their implementation much easier, and will make their maintenance and future extensions much simpler than if a conventional implementation, say in C, C++, or Java, had instead been chosen. Thanks to the high performance of the Maude engine [9]—that can reach 1.66 million free-theory rewrites per second, and from 130 thousand to one million associative-commutative rewrites per second for some applications running on a 500 MHz Alpha—these important benefits in ease of development, understandability, maintainability, extensibility, and in flexibility for introducing formally-defined proof tactics, are achieved without sacrificing performance. Even though they have not been optimized for performance, and in spite of using several levels of reflection and sophisticated rewriting modulo associativity and associativity-commutativity, the tools have quite competitive performance. For example, the inductive theorem prover is very responsive in interactive mode, with input/output time typically dominating over computation time. The Church-Rosser checker also offers reasonable performance. For example, generating all the critical pairs and membership assertions for the number hierarchy from the natural to the rational numbers takes 2,091,898 rewrites, which are performed in 12 seconds running on a 450MHz Pentium II.

The rest of the paper is as follows. We first give a brief review of membership equational logic, rewriting logic, and Maude, including reflective features and the related topic of strategies; and we summarize the reflective design of the tools. Then, each of the tools, including its inference system and its corresponding Maude implementation, is explained and illustrated with examples in a separate section. We end the paper with some concluding remarks.

2. Basic Concepts and Tool Design

2.1. Membership Equational Logic

Membership equational logic [37,3], is conservative extension of both order-sorted equational logic and partial equational logic with existence equations [37]. It supports partiality, subsorts, operator overloading, and error specification.

A *signature* in membership equational logic is a triple $\Omega = (K, \Sigma, S)$ with K a set of *kinds*, (K, Σ) a many-sorted (although it is better to say "many-kinded") signature, and $S = \{S_k\}_{k \in K}$ a K-kinded set of *sorts*.

An Ω-*algebra* is then a (K, Σ)-algebra A together with the assignment to each sort $s \in S_k$ of a subset $A_s \subseteq A_k$. Intuitively, the elements in sorts are the good, or correct, or

nonerror, or defined, elements, whereas the elements without a sort are error or undefined elements.

Atomic formulas are either Σ-equations, or *membership assertions* of the form $t : s$, where the term t has kind k and $s \in S_k$. General sentences are Horn clauses on these atomic formulae, quantified by finite sets of K-kinded variables. That is, they are either conditional equations

$$(\forall X)\ t = t \ \ if \ \ (\bigwedge_i u_i = v_i) \wedge (\bigwedge_j w_j : s_j)$$

or *membership axioms* of the form

$$(\forall X)\ t : s \ \ if \ \ (\bigwedge_i u_i = v_i) \wedge (\bigwedge_j w_j : s_j).$$

Membership equational logic has all the usual good properties: soundness and completeness of appropriate rules of deduction, initial and free algebras, relatively free algebras along theory morphisms, and so on [37].

2.2. Rewriting Logic

Rewriting logic [32] is like a two-faced Janus, in that it expresses an essential equivalence between logic and computation in a particularly simple way. Namely, system *states* are in bijective correspondence with *formulas* (modulo whatever structural axioms are satisfied by such formulas: for example, modulo the associativity and commutativity of a certain connective) and concurrent *computations* in a system are in bijective correspondence with *proofs* (modulo appropriate notions of equivalence between computations and between proofs). Given this equivalence between computation and logic, a rewriting logic axiom of the form $t \longrightarrow t'$ has two readings. Computationally, it means that a fragment of a system's state that is an instance of the pattern t can *change* to the corresponding instance of t' concurrently with any other state changes; that is, the computational reading is that of a *local concurrent transition*. Logically, it just means that we can derive the formula t' from the formula t; that is, the logical reading is that of an *inference rule*.

Rewriting logic is entirely neutral about the structure and properties of the formulas/states t. They are entirely *user-definable* as an algebraic data type satisfying certain equational axioms, so that rewriting deduction takes place *modulo* such axioms. Because of this ecumenical neutrality, rewriting logic has, from a logical viewpoint, good properties as a *logical framework* [31], in which many other logics can be naturally represented. And, computationally, it has also good properties as a *semantic framework* [35], in which many different system styles and models of concurrent computation can be naturally expressed without any distorting encodings.

The design of the present tools exploits and illustrates the logical framework capabilities of rewriting logic by formally specifying the *inference system* of each tool as a rewrite theory. The fact that these tools reason about *equational theories* is not a restriction of the general framework capabilities: inference systems for reasoning about specifications in any other logic could be represented just as easily. However, the fact that rewriting logic contains equational logic as a sublanguage and that Maude has good support for its equational sublanguage is an added advantage.

We can represent a *rewrite theory* as a four-tuple $T = (\Omega, E, L, R)$, where (Ω, E) is a theory in membership equational logic[1] [37,3], L is a set of labels, to label the rules, and R is the set of labeled rewrite rules axiomatizing the local state transitions of the system. Some of the rules in R may be conditional [32].

2.3. Maude

Maude [9,12] is a language whose modules are theories in rewriting logic. The most general Maude modules are called *system modules*. They have the syntax `mod` T `endm` with T the rewrite theory in question, expressed with a syntax quite close to the corresponding mathematical notation[2]. The equations E in the equational theory (Ω, E) underlying the rewrite theory $T = (\Omega, E, L, R)$ are presented as a union $E = A \cup E'$, with A a set of *equational axioms* introduced as *attributes* of certain operators in the signature Ω—for example, an operator $+$ can be declared associative and commutative by keywords `assoc` and `comm`—and where E' is a set of equations that are assumed to be Church-Rosser and terminating *modulo* the axioms A. Maude supports rewriting modulo different combinations of such equational attributes: operators can be declared associative, commutative, with identity, and idempotent [9]. Maude contains a sublanguage of *functional modules* of the form `fmod` (Ω, E) `endfm`, where, as before, $E = A \cup E'$, with E' Church-Rosser and terminating modulo A. A system module `mod` T `endm` specifies the initial model of the rewrite theory T. Similarly, a functional module `fmod` (Ω, E) `endfm` specifies the initial algebra of the equational theory (Ω, E), which is at the same time the initial model of (Ω, E) when viewed as a (degenerate) rewrite theory.

The syntax of Maude modules follows quite closely the mathematical notation for specifying the corresponding mathematical theories. For example, the following module `NAT` specifies the natural numbers in Peano notation.

```
fmod NAT is
  sorts Zero Nat .
  subsort Zero < Nat .
  op 0 : -> Zero [ctor] .
  op s_ : Nat -> Nat [ctor] .
  ops (_+_) (_*_) : Nat Nat -> Nat [comm] .
  vars N M : Nat .
  eq 0 + N = N .
  eq s N + M = s (N + M) .
  eq 0 * N = 0 .
  eq s N * M = M + (N * M) .
endfm
```

Note that the addition and multiplication operators have been declared commutative with the `comm` equational attribute. The `ctor` attribute declares zero and successor as

[1] More generally, the choice of the equational logic underlying rewriting logic is a parameter that can be instantiated to different variants.

[2] See [9] for a detailed description of Maude's syntax, which is quite similar to that of OBJ3 [25]. In this paper we adopt an optimistic viewpoint, assuming that the syntax of the modules that we present is for the most part self-explanatory.

constructors, stating that all other ground terms can be proved equal to terms built using only these two operators.

2.4. Reflection and the META-LEVEL

Rewriting logic is reflective [13,14] in the precise sense that there is a finitely presented rewrite theory U such that for any finitely presented rewrite theory T (including U itself) we have the following equivalence

$$T \vdash t \longrightarrow t' \iff U \vdash \langle \overline{T}, \overline{t} \rangle \longrightarrow \langle \overline{T}, \overline{t'} \rangle,$$

where $\overline{T}$ and $\overline{t}$ are terms representing T and t as data elements of U. Since U is representable in itself, we can achieve a "reflective tower" with an arbitrary number of levels of reflection, since we have

$$T \vdash t \to t' \Leftrightarrow U \vdash \langle \overline{T}, \overline{t} \rangle \to \langle \overline{T}, \overline{t'} \rangle \Leftrightarrow U \vdash \langle \overline{U}, \overline{\langle \overline{T}, \overline{t} \rangle} \rangle \to \langle \overline{U}, \overline{\langle \overline{T}, \overline{t'} \rangle} \rangle \ldots$$

For efficiency reasons, the Maude implementation provides key features of the universal theory U in a built-in module called `META-LEVEL`. In particular, `META-LEVEL` has sorts `Term` and `Module`, so that the representations $\overline{t}$ and $\overline{T}$ of a term t and a module T have sorts `Term` and `Module`, respectively.

Rather than giving here a detailed explanation of the representations $\overline{t}$ and $\overline{T}$ of terms and modules—for which we refer the reader to [8]—we instead illustrate them with simple examples. In the representation of terms, constants are represented as pairs using the operator `{_}_`, with the first argument the constant, in quoted form, and the second argument the sort of the constant, also in quoted form. For example, the constant `0` in the module `NAT` in Section 2.3 is represented as `{'0}'Nat`. The operator `_[_]` corresponds to the recursive construction of terms out of subterms, with the first argument the top operator in quoted form and the second argument the list of its subterms separated by commas. For example, the term `s s 0 + s 0` in the module `NAT` is metarepresented as

```
'_+_['s_['s_[{'0}'Nat]], 's_[{'0}'Nat]].
```

The representation of modules follows very closely Maude's syntax, except that terms in equations, memberships, and rules, are now metarepresented as explained above. For example, the representation $\overline{\texttt{NAT}}$ of the module `NAT` is the following term of sort `Module`:

```
fmod 'NAT is
  nil
  sorts 'Zero ; 'Nat .
  subsort 'Zero < 'Nat .
  op '0 : nil -> 'Zero [none] .
  op 's_ : 'Nat -> 'Nat [none] .
  op '_+_ : 'Nat 'Nat -> 'Nat [ctor] .
  op '_*_ : 'Nat 'Nat -> 'Nat [ctor] .
  var 'N : 'Nat . var 'M : 'Nat .
  none
  eq '_+_[{'0}'Nat, 'N] = 'N .
```

```
  eq '_+_['s_['N], 'M] = 's_['_+_['N, 'M]] .
  eq '_*_[{'0}'Nat, 'N] = {'0}'Nat .
  eq '_*_['s_['N], 'M] = '_+_['M, '_*_['N, 'M]] .
endfm
```

META-LEVEL has also a number of important metalevel functions [9]. For our purposes here we focus on the functions meta-reduce and meta-apply.

The function meta-reduce takes as arguments the representation of a module T and the representation of a term t or of a membership assertion $t : s$ in that module. When the second argument is the representation $\bar{t}$ of a term t in T, the function meta-reduce returns the representation of the fully reduced form of the term t using the equations in T. When the second argument of meta-reduce is the representation of a membership assertion $t : s$, the term t is fully reduced using the equations in T and then the representation of the value of the resulting membership predicate is returned.

The operation meta-apply has the form: meta-apply(T,t,l,σ,n). Its first four arguments are representations in META-LEVEL of a module T, a term t in T, a label l of some rule in T, and a set of assignments (possibly empty) defining a partial substitution σ for the variables in those rules. The last argument is a natural number n. meta-apply is evaluated as follows:

1. the term t is first fully reduced using the equations in T;
2. the resulting term is matched against all rules with label l partially instantiated with σ, with matches that fail to satisfy the condition of their rule discarded;
3. the first n successful matches are discarded; if there is an $(n + 1)$th match, its rule is applied using that match and the steps 4 and 5 below are taken; otherwise {error*, none} is returned;
4. the term resulting from applying the given rule with the $(n + 1)$th match is fully reduced using the equations in T;
5. the pair formed using the constructor {_,_} whose first element is the representation of the resulting fully reduced term and whose second element is the representation of the match used in the reduction is returned.

2.5. Strategies

System modules in Maude are rewrite theories that do not need to be Church-Rosser and terminating. We need then to have flexible ways of controlling the rewriting inference process—which in principle could go in many undesired directions—by means of adequate *strategies*. In Maude, thanks to its reflective capabilities, strategies can be made *internal* to the logic. That is, they can be defined by rewrite rules in a normal module in Maude, and can be reasoned about as with rules in any other module.

In fact, there is great freedom for defining many different strategy languages inside Maude. This can be done in a completely user-definable way, so that users are not limited by a fixed and closed strategy language. The idea is to use the operations meta-reduce and meta-apply as basic strategy expressions, and then to extend the

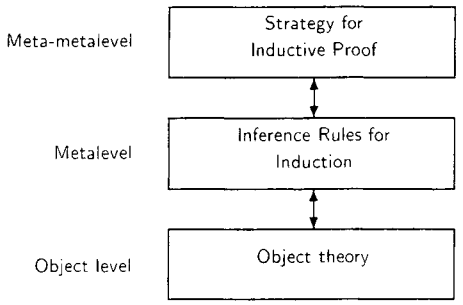


Figure 1.1. The design of the inductive theorem prover.

module `META-LEVEL` by additional strategy expressions and corresponding semantic rules. That is, `meta-reduce` and `meta-apply` allows us to take *elementary steps* in the deduction process, but we can take *bigger steps* by defining more complex *strategy expressions* in a module extending `META-LEVEL` by *equations* that define in rewriting logic the precise rewriting semantics for each of the constructors building up strategy expressions in our strategy language of choice. A number of strategy languages that have been defined following this methodology can be found in [7,9].

2.6. Reflective Design of the Tools

Based on these concepts, each of the tools has then a very simple design. Consider, for example, the design of the inductive theorem prover. Its purpose is to prove inductive properties of a functional module T in Cafe or Maude, which has an initial algebra semantics. The theory T about which we want to prove inductive properties is at the object level. The rules of inference for induction can be naturally expressed as a rewrite theory $\mathcal{I}$. But since this rewrite theory uses T as a data structure—that is, it actually uses its representation $\overline{T}$—the theory $\mathcal{I}$ should be defined at the metalevel. Proving an inductive theorem for T corresponds to applying the rules in $\mathcal{I}$ with some *strategy*. But since the strategies for any rewrite theory belong to the metalevel of such a theory, and $\mathcal{I}$ is already at the metalevel, we need *two levels of reflection* above the object level to clearly distinguish levels and make our design entirely *modular*, so that, for example, we can change the strategy without any change whatsoever to the inference rules in $\mathcal{I}$. This is illustrated by the picture in Figure 1.1.

The inductive Church-Rosser checker tool has a similar reflective design. Again, the module T that we want to check is at the object level. An inference system $\mathcal{C}$ for checking the Church-Rosser property uses $\overline{T}$ as a data structure and therefore is a rewrite theory at the metalevel. Since the checking process can be described in a purely functional way, there is no need in this case for an additional strategy layer at the meta-metalevel: one reflective level suffices.

The reflective design of the tools includes also the design in Maude of a user interface for each tool. For example, the fact that two levels of reflection are used in the inductive theorem prover should be hidden from the user, so that he/she can interact with the prover by entering modules and giving commands at the object level and can get appropriate results and feedback from the tool also at that level. Suitable user interfaces for each tool that accomplish this have been built in Maude using the meta-parsing and meta-pretty-printing functions in `META-LEVEL`, together with a simple `LOOP-MODE` module in Maude [9] that provides input/output. Using these facilities, the interfaces are then built for each tool by specifying in Maude with rewrite rules the desired behavior in response to the different commands that can be entered by the user.

Our approach for building these interfaces has been to integrate the tools within the Full Maude environment [9,18], a language extension of Maude written in Maude that supports parameterized modules, parameter theories, views, and module expressions, and that already has a user interface built using reflective methods. In this way, we have been able to reuse much of the functionality required for the tools' interfaces. Furthermore, in this way we can integrate Full Maude with the tools and submit to them modules in Full Maude's database. We do not give details about the implementation of the interfaces, nor about the integration of the tools with Full Maude. Complete explanations about how this has been done for the Church-Rosser checker can be found in [18].

3. The Inductive Theorem Prover

In this section we present the specification in Maude of an inductive theorem prover to prove theorems about the initial model of a functional module in Cafe or in Maude[3]. As in the case of the Church-Rosser checker tool that will be discussed in Section 4, it is very natural to adopt a reflective design to specify in Maude an inductive theorem-prover. The idea is that the functional module T about which we want to prove inductive theorems is at the object level; an inference system $\mathcal{I}$ for inductive proofs uses $\overline{T}$, the metarepresentation of module T, as data, and therefore can be specified as a rewrite theory at the metalevel; then, different *proof tactics* to guide the application of the rewrite rules that specify the inference rules in $\mathcal{I}$ are strategies that can be represented at the meta-metalevel.

We first introduce an inference system for inductive proofs, and we explain how it can be specified in Maude as a rewrite theory at the metalevel. Then, we discuss the representation of proof tactics at the meta-metalevel and illustrate their use. We also give an inductive proof method to ensure that a set of constructors generate the initial algebra of the specification—so that only terms involving such constructors need to be included in the *test sets* used in the induction scheme—and illustrate this proof method with examples. We also discuss the generation of the test sets themselves.

3.1. Inductive Inference Rules

The following rules 1–6 provide a sound inference system for proving that a universally quantified atomic formula of the form $(\forall X)t = t'$, or of the form $(\forall X)t : s$ is an inductive

[3]Cafe functional modules are order-sorted equational theories; Maude functional modules are theories in membership equational logic. Thanks to the conservative embedding of order-sorted equational logic into membership equational logic [37], Cafe modules are easily translated into corresponding Maude modules.

consequence of a given membership algebra specification. Note the "backwards reasoning" form of these rules. That is, in each rule the inference above the bar is in fact the conclusion that can be established if we can prove the "subgoals" under the bar. This presentation of the rules agrees with the intended use of the inductive prover, in which a user will submit an atomic sentence to be proved as the initial goal, and then this goal will be successively transformed by rewriting—using the inference rules 1–6 as rewrite rules—into different sets of subgoals, until no subgoals are left if the proof succeeds.

Note also that the atomic sentences that these rules can attempt to prove are precisely those of membership equational logic. Since a very general version of order-sorted algebra can be embedded in membership algebra in a conservative way [37], they remain sound for proving inductive theorems of order-sorted functional specifications in Cafe. In fact, the extra generality of membership equational logic allows us to prove useful properties of order-sorted specifications that cannot be stated as order-sorted sentences. Two good examples are sufficient generation proofs—that are expressed as inductive proofs of membership assertions for the defined symbols (see Section 3.3)—and inductive proofs of membership assertions generated as proof obligations by the Church-Rosser Checker to ensure the ground-Church-Rosser property for a given order-sorted specification (see Section 4.1).

In what follows, $vars(t)$ denotes the set of variables that occur in a term t.

1. **Test Set Induction**

$$\frac{M \vdash_{\text{ind}} (\forall X).p}{M \vdash_{\text{ind}} step(p, X \setminus \{x\}, x, s, u_1) \cdots M \vdash_{\text{ind}} step(p, X \setminus \{x\}, x, s, u_n)}$$

where $(\forall X).p$ is an atomic sentence in membership equational logic, $x \in X$ has sort s and $\{u_1, \ldots, u_n\}$ is the test set for sort s and

$$step(p, X, x, s, u) = (\forall vars(u)).\left(\left(\bigwedge_{y:s' \in vars(u),\ s' \leq s} (\forall X).p[y/x]\right) \Rightarrow (\forall X).p[u/x]\right)$$

2. **Proof in Variety**

$$\frac{M \vdash_{\text{ind}} p}{M \vdash p},$$

where p is a sentence in membership equational logic.

3. **Constants Lemma**

$$\frac{M \vdash (\forall \{x_1, \ldots, x_n\}).p}{M \cup \{\texttt{op}\ c_1 : \texttt{->}\ s_1. \cdots \texttt{op}\ c_n : \texttt{->}\ s_n.\} \vdash p[c_1/x_1, \ldots, c_n/x_n]}$$

where $\forall \{x_1, \ldots, x_n\}).p$ is a sentence in membership equational logic, x_i has sort s_i, for $1 \leq i \leq n$, and $c_1, \ldots, c_n$ do not occur in M.

4. **Implication Elimination**

$$\frac{M \vdash (\bigwedge_i (\forall X_i).u_i = v_i) \wedge (\bigwedge_j (\forall X_j).w_j : s_j) \Rightarrow (\forall X).t = t'}{M \cup \{\mathtt{eq}\ u_i = v_i.\}_i \cup \{\mathtt{mb}\ w_j : s_j.\}_j \vdash (\forall X).t = t'}$$

$$\frac{M \vdash (\bigwedge_i (\forall X_i).u_i = v_i) \wedge (\bigwedge_j (\forall X_j).w_j : s_j) \Rightarrow (\forall X).t : s}{M \cup \{\mathtt{eq}\ u_i = v_i.\}_i \cup \{\mathtt{mb}\ w_j : s_j.\}_j \vdash (\forall X).t : s}$$

where in both rules we assume that for each $i \in \{1, \ldots, n\}$, $(vars(u_i) \cup vars(v_i)) \subseteq X_i$, and for each $j \in \{1, \ldots, m\}$, $vars(w_j) \subseteq X_j$.

5. **Proof by Rewriting for Ground Atoms**

$$\frac{M \vdash t = t'}{t \downarrow_M \equiv t' \downarrow_M} \quad \text{if } t,\ t' \text{ are ground.}$$

$$\frac{M \vdash t : s}{t \downarrow_M : s} \quad \text{if } t \text{ is ground.}$$

6. **Equational and Membership Lemmas**

$$\frac{M \vdash_{\mathsf{ind}} p}{M \vdash_{\mathsf{ind}} (\forall X).t = t' \qquad M \cup \{\mathtt{eq}\ t = t'.\} \vdash_{\mathsf{ind}} p}$$

$$\frac{M \vdash_{\mathsf{ind}} p}{M \vdash_{\mathsf{ind}} (\forall X).t : s \qquad M \cup \{\mathtt{mb}\ t : s.\} \vdash_{\mathsf{ind}} p}$$

where $(\forall X).t = t'$, $(\forall X).t : s$, and p are atomic sentences in membership equational logic.

The explicit induction style of the above rules of inference is similar to that used in other explicit induction approaches in equational theorem proving such as those of OBJ [22,23] and LARCH [26].

We give now an overview of the specification in Maude of the inference system just introduced. This specification is given by a module called `ITP`, in which membership equational formulas α are represented as terms $\overline{\alpha}^f$ using the operations `trueFormula`,`equality`, `sortP`, `implication`, `conjunction`, and `VQuantification`. We illustrate this representation using the module `NAT` in Section 2.3. For example, the universally quantified atomic formula $(\forall\{\mathtt{N},\mathtt{M}\}).\mathtt{N}\ \mathtt{+}\ \mathtt{M} = \mathtt{M}\ \mathtt{+}\ \mathtt{N}$ is represented in `ITP` by the term

```
VQuantification((N ; M), equality(N + M, M + N)).
```

Similarly, $(\forall\{\mathtt{N},\mathtt{M}\}).\mathtt{N}\ \mathtt{+}\ \mathtt{M} : \mathtt{Nat}$ is represented by the term

```
VQuantification((N ; M), sortP(N + M, Nat).
```

Sets of terms are built with the constructor `termSet`, with `emptyTermSet` the empty set of terms. The constructor `test` is used to represent the test set for a given sort. A set of test sets is built with the constructor `testSet`, with `emptyTestSet` the empty set

of test sets. For example, the term `test(`$\overline{\texttt{Nat}}$`, termSet(`$\overline{\texttt{0}}$`, `$\overline{\texttt{s N}}$`))` is the representation of the test set[4] for the sort `Nat` in the module `NAT` in Section 2.3.

The (sub)goals for our inductive theorem prover are represented with the constructors `proveinInitial` and `proveinVariety`, for proofs in the initial model and proofs in the variety, respectively. For example, the goal

$$\texttt{NAT} \vdash_{\text{ind}} (\forall\{\texttt{N},\texttt{M}\}).\texttt{N + M} = \texttt{M + N}$$

is represented in ITP by the term

$$\texttt{proveinInitial}(I,\ \overline{\texttt{NAT}},\ \overline{(\forall\{\texttt{N},\texttt{M}\}).\texttt{N + M} = \texttt{M + N}}^{f},\ \overline{\texttt{NATTS}}),$$

where I is a string of positive numbers, and $\overline{\texttt{NATTS}}$ is the representation of the set of test sets for the sorts in the functional module `NAT`. The strings of positive numbers are used to number the (sub)goals in a proof. Finally, sets of (sub)goals are built with the constructor `goalSet`, with `emptyGoalSet` the empty set of goals.

The rewrite rules in the module `ITP` mirror very closely the inference system described above. Here we only explain the specification of the inference rules **Implication Elimination** and **Proof by Rewriting for Ground Atoms**. When explaining the rules in `ITP` we also explain briefly the auxiliary functions used in them. Note that our specification makes use of the reflective capabilities of Maude, in particular: (i) that functional modules can be treated as data that can be manipulated; and (ii) that reducing a term to its normal form is reified by the built-in function `meta-reduce`. Full details about the specification of the ITP tool can be found in [10].

We start with the declaration of some variables that will be used in the rules:

```
var IS : IntString .   vars T1 T2 : Term .  var Th : FModule .
vars Alpha Beta : Formula .  var G : GoalSet .
```

The inference rules for **Implication Elimination** are jointly specified by the rule `implicationElimination`.

```
op addToModule : Formula Module -> Module .

rl [implicationElimination]:
  goalSet(proveinVariety(IS, Th, implication(Alpha, Beta)), G)
  => goalSet(proveinVariety(IS, addToModule(Alpha, Th), Beta), G) .
```

Let T be a functional module, and $\alpha = (\forall X_1).p_1 \wedge \ldots \wedge (\forall X_n).p_n$ a sentence over the signature of T such that, for $1 \leq i \leq n$, p_i is either $u_i = v_i$ or $w_i : s_i$. Then, the function `addToModule(`$\overline{\alpha}^{f}$`,` $\overline{T}$`)` returns $\overline{T'}$, where T' is the module T extended in the following way:

[4]The correctness of a test set for a given sort depends on having shown that the subsignature of constructors on which the test set is based does indeed generate all terms of that sort, in the sense that any term of that sort is provably equal to a constructor term. As explained in Section 3.3, given a subsignature of constructors for our specification we can use the inductive theorem prover to show that indeed the constructors generate all terms of all sorts, and then we can mechanically check or generate test sets for each sort. In what follows we shall assume that this has already been done for the specification in question.

for each component $(\forall X_i).u_i = v_i$ in the conjunction α the equation `eq` $u_i = v_i$ is added to the set of equations of T; and for each component $(\forall X_i).w_i : s_i$ in the conjunction α the membership axiom `mb` $w_i : s_i$ is added to the set of membership axioms of T.

The inference rules for **Proof by Rewriting for Ground Atoms** are specified by the rules `byRewriting`. Notice the use of the built-in function `meta-reduce` in our specification of these inference rules.

```
rl [byRewriting]:
  goalSet(proveinVariety(IS, Th, equality(T1, T2)), G)
  => goalSet(proveinVariety(IS, Th,
             equal(meta-reduce(Th, T1), meta-reduce(Th, T2))), G) .
rl [byRewriting]:
   goalSet(proveinVariety(IS, Th, sortP(T1, S)), G)
  => goalSet(
        proveinVariety(IS, Th,
           equal(meta-reduce(Th, T1 : S), {'true}'Bool)), G) .

eq proveinVariety(IS, Th, equal(T, T)) = emptyGoalSet .
```

3.2. Proof Strategies

As explained above, different *proof tactics* to guide the applications of the rewrite rules in the module `ITP` are strategies that can be represented at the meta-metalevel.

In [10], two proof tactics are defined in detail: a strategy `simplify` that tries to simplify a particular (sub)goal as much as possible, and a strategy `induction` that, when there is only one goal to be proved, applies the induction rule on a selected variable and then simplifies the resulting subgoal as much as possible. These strategies are computed by a function `rewInWith` that takes three arguments: the representations of a module T, the representation of a term t, and the strategy to be computed. The definition of `rewInWith` is such that, as the computation of a given strategy proceeds, t gets rewritten by controlled application of rules in T, and the strategy S is rewritten into the remaining strategy to be computed; in case of termination, this is the `idle` strategy.

As an example, we can use the strategy `induction` to prove the associativity of the operation `+` (addition) as defined in the module `NAT` in Section 2.3, that is, the formula $(\forall\{\mathtt{X},\mathtt{Y},\mathtt{Z}\}).\mathtt{X} + (\mathtt{Y} + \mathtt{Z}) = (\mathtt{X} + \mathtt{Y}) + \mathtt{Z}$. We prove it computing the `induction` strategy on the variable `Z`, that is, the term

```
rewInWith(ITP,
 proveinInitial(1, NAT, (∀{X,Y,Z}).X+(Y+Z) = (X+Y)+Z^f , NATTS),
 induction(Z)) .
```

which is reduced to

```
rewInWith(ITP, emptyGoalSet, idle) .
```

3.3. Sufficient Generation

The test set induction rule instantiates an induction variable by the terms in a test set. The most classical case is the instantiation of a variable of sort `Nat` by `0` and `s N`. These expressions suffice, because zero and successor are the *constructors* that build up all natural numbers. In general, however, how can we justify the correctness of a test set? We need to show that all the ground terms of a given sort are provably equal to constructor terms that are instances of the terms in the test set for that sort. This brings us to the topic of sufficient generation, that is, of showing that a given constructor subspecification—that may actually satisfy some equations of its own—generates all the sorts in the initial algebra of our specification, in the sense that all terms of that sort can be proved equal to constructor terms. We give here a general method of using the inductive theorem prover to prove sufficient generation, and illustrate the method with some examples; then we discuss a simple test set generation scheme. Our method is based on a theorem about protecting extensions of membership equational logic specifications by Bouhoula, Jouannaud, and Meseguer [3]. For the sake of a simpler exposition, we deal here with the case of order-sorted specifications. The more general case of membership equational logic specifications has a similar treatment and will be discussed elsewhere.

Sufficient generation is a property somewhat weaker than sufficient completeness. Given an order-sorted specification $T = (S, \leq, \Sigma, \Gamma)$ and a subspecification of constructors $T_0 = (S, \leq, \Omega, \Gamma_0)$, with $T_0 \subseteq T$, we say that the initial algebra of T is *sufficiently complete* relative to T_0 if the unique Ω-homomorphism

$$\mathcal{T}_{T_0} \longrightarrow \mathcal{T}_T|_\Omega,$$

from the initial algebra of T_0 to the Ω-algebra obtained from the initial algebra of $\mathcal{T}_T$ by forgetting the operations in $\Sigma - \Omega$, is an *isomorphism*. Instead, we say that the initial algebra of T is *sufficiently generated* by the subsignature of constructors Ω, if the unique Ω-homomorphism

$$\mathcal{T}_\Omega \longrightarrow \mathcal{T}_T|_\Omega$$

is surjective for each sort $s \in S$. This is equivalent to requiring that for each $s \in S$ and each $t \in \mathcal{T}_{\Sigma,s}$ there is a $t_0 \in \mathcal{T}_{\Omega,s}$ with

$$T \vdash t = t_0.$$

Sufficient completeness is of course a stronger property than sufficient generation. But for being justified in using only constructor terms in the test sets of an inductive proof, only sufficient generation must be checked.

We give below a general method for checking that the initial algebra of T is sufficiently generated by a subsignature of constructors Ω. The advantage of this method is that we can use the inductive theorem prover to check the proof obligations that it generates.

The method is as follows. Given T and a subsignature of constructors Ω, we define a new specification $E_\Omega T = (ES, \leq_E, E\Sigma, \Gamma)$ with:

- sorts ES those of S plus, for each connected component C of $(S, \leq)$ a new "error sort" E_C greater than all the sorts in C,

- order $\leq_E$ the obvious extension of $\leq$ to make each E_C the top of each component,
- operators $E\Sigma$ those of Ω (with same arity and coarity), plus for each operator $f : s_1 \dots s_n \to s$ in $\Sigma - \Omega$ with $c(s_1), \dots, c(s_n), c(s)$ the corresponding connected components, an operator $f : E_{c(s_1)} \dots E_{c(s_n)} \to E_{c(s)}$, and
- axioms Γ exactly the same as those of T.

Then, to prove that the initial algebra of T is sufficiently generated by Ω, it is enough to show that for each[5] $f : s_1 \dots s_n \to s$ in $\Sigma - \Omega$,

$$E_\Omega T \vdash_{\text{ind}} f(x_1, \dots, x_n) : s \qquad (\ddagger)$$

where the variables $x_1, \dots, x_n$ have respective sorts $s_1, \dots, s_n$. Note that this inductive requirement takes place in membership equational logic. Therefore, we implicitly use the conservative embedding of order-sorted equational logic into membership equational logic [37], and the fact that Maude's functional modules are theories in membership equational logic.

We can then use the inductive theorem prover to prove each of the proof obligations ($\ddagger$). Note that, since the specification $E_\Omega T$ is of course sufficiently generated by its own signature $E\Sigma$, and $E\Sigma$ has the property that for each $s \in S$ all operators of coarity less or equal to s are in Ω, when proving the proof obligations ($\ddagger$), we are justified in using only Ω-*terms* in the test sets of sorts $s \in S$ in $E_\Omega T$.

Test Set Generation

We illustrate the use of the general methodology to prove sufficient generation below. However, since for proving inductively sufficient generation we need a test set, we explain now a simple test set generation scheme.

Once we have proved that a subsignature of constructors sufficiently generates the initial algebra of a specification, we are justified in using only those constructors to associate to each sort a test set. Of course, the choice of what terms to use for a test set is not unique. For example, in a specification of the natural numbers with a sort `NzNat` of nonzero natural numbers and constructors `0 : -> Nat`, and `s : Nat -> NzNat`, we could use as test set for the sort `NzNat` the term `s N` with `N` a variable of sort `Nat`, but we could also have used as test set of sort `NzNat` the terms `s 0` and `s s N`, with `N` a variable of sort `Nat` as well.

Our test set generation function chooses the first, simpler alternative. That is, it always yields terms of depth zero or one in the test set of each sort. Such terms are obtained by inspecting what constructors have sort smaller or equal to the sort in question; for each such constructor $\mathtt{f} : \mathtt{s}_1 \dots \mathtt{s}_n \to \mathtt{s}$ a term $\mathtt{f}(\mathtt{x}_1, \dots, \mathtt{x}_n)$ with $\mathtt{x}_i$: $\mathtt{s}_i$, for $1 \leq i \leq n$, is then added to the test set. As an optimization that avoids unnecessary subgoals in inductive proofs, if a constructor has two overloadings, with the rank of one of them smaller or equal to that of the other, and both with coarities smaller or equal to the sort in question, then only the most general constructor is used in generating the test set for that sort.

[5]Note that an operator declaration $f : s_1 \dots s_n \to s$ can be in $\Sigma - \Omega$ while at the same time another declaration $f : s'_1 \dots s'_n \to s'$ can be in Ω. For example, `s : Int -> Int` is not a constructor, but `s : Nat -> Nat` is a constructor.

Given a subsignature of constructors for which sufficient generation has been proved, the function `genTestSets` takes a module[6] and generates the test sets for each sort in it.

The function that generates the test sets is `genTestSets`, which takes a module and returns a pair of sort `TestSetSol` consisting of a module and the family of test sets generated.

```
op genTestSets : Module -> TestSetSol .
```

It may be that in the process of generating the test sets new variables have to be added to the original specification. Therefore, the test sets generated have to be used together with this new module.

Proving Sufficient Generation of Natural Numbers

We shall prove that the constructors for the module `NAT` presented in Section 2.3 are `0` and `s_`. To do this we create a module `ENAT` as a copy of `NAT` in which we introduce a new sort `ErrorNat` as supersort of `Nat`, and move up to this new sort all the defined operations, except those considered as candidates for constructors, that is, we declare operators `_+_` and `_*_` as

```
ops (_+_) (_*_) : ErrorNat ErrorNat -> ErrorNat [comm] .
```

and leave unchanged the declarations for `0` and `s_`.

The first thing we need in order to be able to use the theorem prover is to obtain a test set for `ENAT`. What we can do is to consider all the operators of `ENAT` as constructors, something we are always justified in doing, since the entire operator set obviously generates the initial algebra. For this purpose we call the function `genTestSets` presented above with a metalevel syntactic variant `CNAT` of the module `ENAT` in which all the operators are syntactically marked as constructors

Then, calling the function `genTestSets` with `CNAT` as argument gives us a pair consisting of a modified module, in which some variables may have been introduced, and the corresponding family of test sets. Transforming back the module returned with the test sets to the original module syntax without constructor declarations we obtain a module which, in order to continue using the same notation introduced above, we shall call `ENAT`.

Let us call `ENATTS` the test sets returned:

```
testSet(test(Zero, 0),
        test(Nat, termSet(0, s N)),
        test(ErrorNat, termSet(0, s N, ErrorNat&'0 + ErrorNat&'1,
                               ErrorNat&'0 * ErrorNat&'1)))
```

Now we can prove the proof obligations for sufficient generation by the method already explained. To prove that the operators `s_` and `0` are constructors for `NAT`, we need to prove inductively that in the module `ENAT`, given variables `N` and `M` of sort `Nat`, `N + M` and `N * M` are of sort `Nat`. A detailed description of the proofs using the inductive theorem prover presented above and some other examples can be found in [10].

[6] As said in Section 2.3, in Maude 1.0.5 constructor opertors can be distinguished using the `ctor` attribute.

4. The Church-Rosser Checker

The goal of *executable* equational specification languages is to make *computable* the abstract data types specified in them by initial algebra semantics. In practice this is accomplished by using specifications that are *ground*-Church-Rosser and terminating, so that the equations can be used from left to right as simplification rules. This approach is fully general: indeed, a well-known result of Bergstra and Tucker [2] shows that *any* computable algebraic data type can be specified by means of a finite set of ground-Church-Rosser and terminating equations, perhaps with the help of some auxiliary functions added to the original signature.

Therefore, for formal reasoning purposes it becomes very important to know whether a given specification is indeed ground-Church-Rosser and terminating. A nontrivial question is how to best support this with adequate tools. For proving termination one can use standard termination proof techniques that are well-supported in tools such as CiME [15]. A thornier issue is what to do for establishing the ground-Church-Rosser property for a terminating specification. The problem is that a specification with an initial algebra semantics can often be ground-Church-Rosser even though some of its critical pairs may not be joinable. That is, the specification can often be ground-Church-Rosser without being Church-Rosser for arbitrary terms with variables. In such a situation, blindly applying a completion procedure that is trying to establish the Church-Rosser property for arbitrary terms may be both quite hopeless—the procedure diverges or gets stuck because of unorientable rules, and even with success may return a specification that is quite different from the original one—and unnecessary: the specification was ground-Church-Rosser!

Our Church-Rosser checker tool is oriented to checking specifications with an initial algebra semantics that have already been proved terminating using a termination tool and now need to be checked to be ground-Church-Rosser. Since, for the reasons mentioned above, user interaction will typically be quite essential, completion is not attempted. Instead, if the specification cannot be shown to be ground-Church-Rosser by the tool, proof obligations are generated and are given back to the user as a useful guide in the attempt to establish the ground-Church-Rosser property. Since this property is in fact inductive, in some cases the inductive theorem prover can be enlisted to prove some of these proof obligations (see Section 4.3). In other cases, the user may in fact have to modify the original specification by carefully considering the information conveyed by the proof obligations. We give in Section 4.3 some methodological guidelines for the use of the tool, and illustrate the use with some examples; we also explain there that the issue of finding general inductive proof techniques for proving the ground-Church-Rosser property is at the moment an interesting open problem.

The present tool only accepts order-sorted conditional specifications such that each of the operation symbols has either no equational attributes, or only the commutativity attribute[7]. Furthermore, it is assumed that such specifications do not contain any built-in

[7] As we shall see in Section 4.2, in the current version of the tool the unification and matching algorithms are specified equationally, and only for signatures whose operators do not have equational attributes, or have just the commutativity attribute. Maude will have built-in matching and unification functions `meta-match` and `meta-unify` in the near future. Once they are available, the tool will be easily extensible

sort or function, and that they have already been proved terminating using another tool. The tool attempts to establish the Church-Rosser property *modulo* the commutativity of some of the operators by checking a sufficient condition. For order-sorted specifications, being Church-Rosser and terminating means not only confluence—so that a unique normal form will be reached—but also a *descent* property, namely that the normal form will have the least possible sort among those of all other equivalent terms. Therefore, the tool's output consists of a set of critical pairs and a set of membership assertions that must be shown, respectively, ground-joinable, and ground-rewritable to a term with the required sort.

After introducing some basic formal concepts and results underlying the tool's design, and after discussing several auxiliary functions used by the tool, including order-sorted commutative unification, we explain key parts of the tool and then give recommendations for its use and some examples.

4.1. Church-Rosser Order-Sorted Specifications

In this section we introduce the notion of Church-Rosser order-sorted specification. We assume specifications of the form $\mathcal{S} = (\Sigma, R \cup A)$ where Σ is a preregular order-sorted signature [24], and $R \cup A$ is a set of equations such that A is a set of *sort-preserving* equational axioms, that is, whenever $t =_A t'$ we have[8] $LS(t) = LS(t')$. The equations R will be used as rewrite rules modulo the axioms A. Furthermore, in what follows, and for the purposes of the present tool, the axioms A will only consist of *commutativity* axioms for certain binary operators in Σ. The problem, then, is to check whether our specification $\mathcal{S}$ has the Church-Rosser property.

After giving some auxiliary definitions, we introduce the notion of Church-Rosser order-sorted specifications, and describe the sufficient condition used by our tool to attempt checking the Church-Rosser property.

The Confluence Property

We shall use the standard notation for the positions of a term [1]; we denote *the subterm of t at position p* as $t|_p$ and a term t with its subterm $t|_p$ replaced by the term t' as $t[t']_p$. Recall that the set of variables occurring in a term t is denoted $vars(t)$. Then, given a substitution $\sigma = \{x_1 \leftarrow t_1, \ldots, x_n \leftarrow t_n\}$, we denote by $D(\sigma)$ the set of variables $\{x_1, \ldots, x_n\}$, and by $I(\sigma)$ the set $\bigcup_{i=1}^{n} vars(t_i)$.

We say that a term t *A-overlaps* another term with distinct variables t' if there is a nonvariable subterm $t'|_p$ of t' for some position p of t' such that the terms t and $t'|_p$ can be A-unified.

Definition 4.1 *Given an order-sorted equational specification $\mathcal{S} = (\Sigma, R \cup A)$ with the above assumptions, and given conditional rewrite rules $l \to r$ `if` $\bar{u}\downarrow \bar{v}$ and $l' \to r'$ `if` $\bar{u'}\downarrow \bar{v'}$ in R such that $vars(l) \cap vars(l') = \emptyset$ and $l|_p\sigma =_A l'\sigma$, for some nonvariable position p of l and A-unifier σ, then the triple*

$$ccp(l\sigma[r'\sigma]_p,\ r\sigma,\ \bar{u}\sigma\downarrow \bar{v}\sigma \wedge \bar{u'}\sigma\downarrow \bar{v'}\sigma)$$

to the case in which the corresponding combinations of equational attributes are supported.

[8] We denote the least sort [24] of a term t by $LS(t)$.

is called a (conditional) critical pair.

We consider all unifiers, not only the most general ones. Note also that the critical pairs accumulate the substitution instances of the conditions in the two rules, as in [3].

Given a specification $\mathcal{S} = (\Sigma, R \cup A)$, a critical pair $ccp(t, t', c)$ is more general than another critical pair $ccp(u, u', c')$ if there exists a substitution σ such that $t\sigma =_A u$, $t'\sigma =_A u'$, and $c\sigma =_A c'$.

Then, given a specification $\mathcal{S}$, let $\mathrm{MCP}(\mathcal{S})$ denote the set of most general critical pairs between rules in $\mathcal{S}$ that, after simplifying both sides of the critical pair using the equations in $\mathcal{S}$, are not identical critical pairs modulo A of the form $ccp(t, t, c)$. To determine the overlaps of the lefthand sides of the rules the variables in them have to be renamed in order to get disjoint sets of variables appearing in each of them. Under the assumption that the order-sorted equational specification $\mathcal{S}$ is terminating, then, if $\mathrm{MCP}(\mathcal{S}) = \emptyset$, we are guaranteed that the specification $\mathcal{S}$ is *confluent*, and therefore, each term t has a unique canonical form $t\downarrow_{\mathcal{S}}$.

The Descent Property

For an order-sorted specification it is not enough to be confluent. The canonical form should also provide the most complete information possible about the sort of a term. This intuition is captured by our notion of Church-Rosser specifications.

Definition 4.2 *We call a confluent and terminating order-sorted specification $\mathcal{S}$* Church-Rosser *iff it additionally satisfies the following* descent *property: for each term t we have $LS(t) \geq LS(t\downarrow_{\mathcal{S}})$. Similarly, we call a ground-confluent and terminating specification $\mathcal{S}$* ground-Church-Rosser *iff for each ground term t we have $LS(t) \geq LS(t\downarrow_{\mathcal{S}})$.*

Note that these notions are more general and flexible than the requirement of confluence and *sort-decreasingness* [29,21]. The issue is how to find checkable conditions for descent that, in addition to the computation of critical pairs, will ensure the Church-Rosser property. This leads us into the topic of specializations.

Given an order-sorted signature $(S, \leq, \Sigma)$, a sorted set of variables X can be viewed as a pair $(\bar{X}, \mu)$ where $\bar{X}$ is a set of variable names and μ is a sort assignment $\mu\colon \bar{X} \to S$. Thus, a *sort assignment* μ for X is a function mapping the names of the variables in $\bar{X}$ to their sorts. The ordering $\leq$ on S is extended to sort assignments by

$$\mu \leq \mu' \Leftrightarrow \forall x \in \bar{X}, \mu(x) \leq \mu'(x).$$

We then say that μ' *specializes* to μ, via the substitution

$$\rho : (x : \mu(x)) \leftarrow (x : \mu'(x))$$

called a *specialization* of $X = (\bar{X}, \mu')$ into $\rho(X) = (\bar{X}, \mu)$. Note that if the set of sorts is finite, or if each sort has only a finite number of sorts below it, then a finite sorted set of variables has a finite number of specializations.

The notion of specialization can be extended to axioms. A specialization of an equation $(\forall X, l = r)$ is another equation $(\forall \rho(X), \rho(l) = \rho(r))$ where ρ is a specialization of X. A

specialization of a rule $(\forall X, l \to r \texttt{ if } c)$ is a rule $(\forall \rho(X), \rho(l) \to \rho(r) \texttt{ if } \rho(c))$ where ρ is a specialization of X.

The checkable conditions that we have to add to the critical pairs to test for the descent property are called membership assertions.

Definition 4.3 *Let $\mathcal{S}$ be an order-sorted specification whose signature satisfies the assumptions already mentioned. Then, the set of* (conditional) membership assertions *for a conditional equation $t = t'$ if c is defined as the set of assertions $t'\theta : LS(t\theta)$ if $c\theta$ such that θ is a specialization of $vars(t)$ and $LS(t'\theta \downarrow_{\mathcal{S}}) \not\leq LS(t\theta)$.*

A membership assertion $t : s$ if c is more general than another membership assertion $t' : s'$ if c' if there exists a substitution σ such that $t\sigma =_A t'$, $s \leq s'$, and $c\sigma =_A c'$.

A fundamental result[9] underlying our tool is that the absence of critical pairs and of membership assertions in such an output is a sufficient condition for a terminating specification $\mathcal{S}$ to be *Church-Rosser*. In fact, for terminating unconditional specifications this check is a necessary and sufficient condition; however, for conditional such specifications, the check is only a sufficient condition because if the specification has conditional equations we can have unsatisfiable conditions in the critical pairs or in the membership assertions; that is, we can have $\langle\ \mathrm{MCP}(\mathcal{S}),\ \mathrm{MMA}(\mathcal{S})\ \rangle \neq \langle\ \emptyset, \emptyset\ \rangle$ with $\mathcal{S}$ still Church-Rosser. Furthermore, even if we assume that the specification is unconditional, since for specifications with an initial algebra semantics we only need to check that $\mathcal{S}$ is ground-Church-Rosser, we may often have specifications that satisfy this property, but for which the tool returns nonempty sets of critical pairs or of membership assertions as proof obligations.

4.2. The Specification of a Church-Rosser Checker

The Church-Rosser checker tool is written entirely in Maude. It is in fact an executable specification in rewriting logic of the inference system to check a sufficient condition for the Church-Rosser property.

The input to the Church-Rosser checker tool is given by the functional module we want to check. As we mentioned in Section 2.6, the tool has been completely integrated within the Full Maude environment [9,18], so that the Church-Rosser property can be checked on any specification satisfying the conditions mentioned above that has been previously entered to Full Maude. Furthermore, the specification in question may be *structured*, with other specifications, perhaps defined by module expressions, as submodules.

After introducing order-sorted unification and matching, we explain the specifications of the confluence and descent checks. The specification of the checking of the Church-Rosser property in then presented.

Order-Sorted Commutative Unification and Matching

In this section we sketch the order-sorted unification algorithm we have implemented. It yields a complete set of unifiers for a unification problem in which it is assumed that the order-sorted signature of the specification in question can have some operators declared to be *commutative*—in which case, all the remaining subsort-overloaded versions of any such

[9] A detailed proof of this result is beyond the scope of this work, and will be presented elsewhere. For related results in membership equational logic see [3].

operator should also have been declared commutative—but no other equational axioms have been declared as attributes for the operators. Therefore, the order-sorted unification is performed *modulo* the commutativity of certain operators.

The basic idea is to turn each of the textbook-style inference rules for such a unification algorithm into corresponding rewrite rules in Maude. Figure 1.2 below gives the inference system; their Maude counterparts can be found in [10]. Here we only summarize the basic concepts and the top-level functionality.

Following [27], we unify using the sort information as soon as possible in order to quickly discard failures. Then, we complete the simplification process and push the assertions of the sorts on the solutions that have been found. A similar approach has been followed for order-sorted matching.

The order-sorted unification algorithm for solving equations (pairs of the form $t =_c^? t'$) can be described as the result of a simplification step followed by a solving step. The simplification step is a sequence of decomposition, merging, and mutation steps, transforming the initial unification problem into an equivalent disjunction of systems of fully decomposed equations of the form $x =_c^? t$, where x is a variable appearing only once in the system. The solving step consists in finding the finite set of solutions for the simplified system.

We give in Figure 1.2 the set of inference rules used for the unification algorithm. They operate on 3-tuples of the form $\langle V;\ E;\ \sigma\rangle$ and on 4-tuples of the form $[V;\ C;\ \sigma;\ \theta]$. A 3-tuple $\langle V;\ E;\ \sigma\rangle$ consists of a set of variables V, a set of equations E, and a substitution σ. A 4-tuple $[V;\ C;\ \sigma;\ \theta]$ consists of a set of variables V, a set of membership constraints C, and substitutions σ and θ. The rules operating on the first kind of tuples correspond to the first stage of the process, which is quite similar to syntactic unification. The main differences are in the rules **Check** and **Eliminate**, in which the sort information is used to try to quickly discard failure. In the second stage the constraints on the solutions are checked. We assume a well-founded order $\succ$ on equations, and we denote by $s \cap s'$ the set of maximal lower bounds of sorts s and s'.

The specification for order-sorted matching is quite similar to the specification for unification. The main differences come from the fact that now we have to consider non-commutative equations, since to find a match from a term t to another term t' is to find a substitution σ such that $t' =_c t\sigma$. The check for the correct sorting of an equation in solved form $x =_c^? t$ is reduced to comparing the sort of the variable x with the least sort of the term t being assigned.

Confluence Check

We present in this section the specification for the confluence check, carried out by the `conflCheck` function. We assume that the module passed as argument to `conflCheck` has been extended with renamed copies of the variables in it. Specifically, we assume that for each variable V in the original module there is a variable V`@` which is not used. Since to check the overlappings there cannot be common variables between the terms, we generate a renamed copy of one of the equations just by renaming each variable.

We declare a sort `CritPair` for critical pairs and a sort `CritPairSet` for sets of critical pairs, and constructors for them. The constructors for critical pairs (`cp`) and for conditional critical pairs (`ccp`) have, respectively, two and four arguments. The two ar-

Deletion of Trivial Equations

$$\frac{\langle\ V;\ \{t =_c^? t,\ E\};\ \sigma\ \rangle}{\langle\ V;\ E;\ \sigma\ \rangle}$$

Check

$$\frac{\langle\ \{x : s,\ V\};\ \{x =_c^? t,\ E\};\ \sigma\ \rangle}{\mathit{failure}}$$

if $x \neq t$ and (x occurs in t or $s \cap LS(t) = \emptyset$)

Decomposition

$$\frac{\langle\ V;\ \{f(t_1, \ldots, t_n) =_c^? f(t'_1, \ldots, t'_n),\ E\};\ \sigma\ \rangle}{\langle\ V;\ \{t_1 =_c^? t'_1, \ldots, t_n =_c^? t'_n,\ E\};\ \sigma\ \rangle}$$

if $n \neq 2$ or f noncommutative

$$\frac{\langle\ V;\ \{f(t_1, t_2) =_c^? f(t'_1, t'_2),\ E\};\ \sigma\ \rangle}{\langle\ V;\ \{t_1 =_c^? t'_1,\ t_2 =_c^? t'_2,\ E\};\ \sigma\ \rangle \vee \langle\ V;\ \{t_1 =_c^? t'_2,\ t_2 =_c^? t'_1,\ E\};\ \sigma\ \rangle}$$

if f commutative

Clash of Symbols

$$\frac{\langle\ V;\ \{f(t_1, \ldots, t_n) =_c^? g(t'_1, \ldots, t'_m),\ E\};\ \sigma\ \rangle}{\mathit{failure}}$$

if $n \neq m$ or $f \neq g$

Merging

$$\frac{\langle\ \{x : s,\ V\};\ \{x =_c^? t,\ x =_c^? t',\ E\};\ \sigma\ \rangle}{\langle\ \{x : s,\ V\};\ \{x =_c^? t,\ t =_c^? t',\ E\};\ \sigma\ \rangle}$$

if $x = t \succ t = t'$

Eliminate

$$\frac{\langle\ \{x : s,\ V\};\ \{x =_c^? t,\ E\};\ \sigma\ \rangle}{\langle\ \{x : s,\ V\};\ E\theta;\ \{\sigma\theta, \theta\}\ \rangle}$$

with $\theta = \{x \leftarrow t\}$

if x does not occur in t and $s \cap LS(t) \neq \emptyset$

Transition

$$\frac{\langle\ V;\ \emptyset;\ \sigma\ \rangle}{[\ V;\ \emptyset;\ \sigma;\ \sigma\ ]}$$

Solving $(x \leftarrow t)$

$$\frac{[\ \{x : s,\ V\};\ C;\ \{x \leftarrow t,\ \sigma\};\ \theta\ ]}{[\ V;\ \{(t : s),\ C\};\ \sigma;\ \theta\ ]}$$

Solving $(x : s)$

$$\frac{[\ \{x : s,\ V\};\ \{x : s',\ C\};\ \emptyset;\ \theta\ ]}{\bigvee_{s'' \in s \cap s'} [\ \{x : s'',\ V\};\ C;\ \emptyset;\ \theta\ ]}$$

Solving $(f(t_1, \ldots, t_n) : s)$

$$\frac{[\ V;\ \{f(t_1, \ldots, t_n) : s, C\};\ \emptyset;\ \theta\ ]}{\bigvee_{\substack{f : s_1 \ldots s_n \to s' \\ s' \leq s \\ s_1 \ldots s_n \ \mathit{maximal}}} [V;\ \{t_1 : s_1, \ldots, t_n : s_n, C\};\ \emptyset;\ \theta]}$$

Figure 1.2. Inference rules for commutative order-sorted unification.

guments of cp and the first two of ccp are the terms forming the critical pair. The last two arguments in a conditional critical pair correspond to the condition, which is given following the conventions for conditions in membership axioms, equations, and rules in the META-LEVEL module.

```
op cp : Term Term -> CritPair .
op ccp : Term Term Term Term -> CritPair .
```

Given a specification $\mathcal{S}$, the critPairs function

```
op critPairs : Module -> CritPairSet .
```

finds all the critical pairs between the equations in $\mathcal{S}$ considered as rules, oriented from left to right. One critical pair is generated for each unifier for each of the possible nonvariable overlappings of the lefthand sides of any two equations in the module. These critical pairs are calculated by finding all the possible such pairs for each of the equations in the module with a renamed copy of each one of the other equations in the module including itself. For each pair of equations, their left sides are unified at any nonvariable position of the term of the first equation, and then a critical pair is constructed for each one of the solutions of the unification problem.

Once all the critical pairs have been generated, the trivial ones—those of the forms cp(t, t) or ccp(t, t, c), for some term t and condition c—are removed by the function delete, which is applied again to the set of critical pairs resulting from the simplification process. This simplification is achieved by the function simplify, which uses the metalevel function meta-reduce to reduce both sides of each of the critical pairs to their normal forms in the given specification. Finally, we apply the function maxCritPairSet to get the set of maximal critical pairs among those remaining. In order to get a smallest possible set of critical pairs, besides eliminating trivial ones, we need to also take care of those that are repeated or that are less general than some other critical pair. The function maxCritPairSet returns the set of most general critical pairs. The conflCheck function is then specified as follows.

```
op delete : CritPairSet -> CritPairSet .
op simplify : CritPairSet Module -> CritPairSet .
op maxCritPairSet : CritPairSet Module -> CritPairSet .
op conflCheck : Module -> CritPairSet .
eq conflCheck(M)
  = maxCritPairSet(delete(simplify(delete(critPairs(M)), M)), M) .
```

Nonconfluence of a Natural Numbers Specification

Our first example illustrating the Church-Rosser checker is the very simple specification NAT of the natural numbers presented in Section 2.3. This specification is perfectly reasonable. Its initial model is the set of natural numbers $\mathbb{N}$, with the sum and product operators. However, although it is ground-confluent it is not confluent. The output given by the tool is as follows.

```
Checking solution :
  var N : Nat .  var N@ : Nat .
  cp s (N + (N@ + (N * N@))) = s (N@ + (N + (N * N@))) .
```

Note that all variables in the critical pairs, and in the membership assertions are collected and included as part of the output. This critical pair was generated from one of the solutions of the unification corresponding to overlapping at the top the lefthand sides of the equation `s N * M = M + (N * M)` and of its renamed copy `s N@ * M@ = M@ + (N@ * M@)`. One critical pair is generated for each of the solutions of the unification of both lefthand sides at the top. After eliminating the trivial one and reducing each of the terms of the other one in the specification we obtain the critical pair returned by the tool.

In this case the set of membership assertions is empty. This means that the equations are descending. We shall study descent checking in below, and will return to this example, to make it confluent, in Section 4.3.

Descent Check

Given a specification $\mathcal{S}$, `descCheck` returns $\mathrm{MMA}(\mathcal{S})$, that is, the set of most general membership assertions among those generated for all specializations of the equations not satisfying the descent condition. For each one of the possible specializations of each of the equations in the module the least sort of the term in the lefthand side is compared with the least sort of the term in the righthand side, reduced to its normal form. Thus, for each equation $(\forall X, l = r)$ that, considered as a rewrite rule, fails to satisfy the descent condition, that is, $LS(r\downarrow) \not\leq LS(l)$, a membership assertion of the form `mb` r `:` $LS(l)$ is generated. Similarly, for a conditional equation $(\forall X, l = r$ `if` $c)$ failing to satisfy the descent condition, a *conditional* membership assertion of the form `cmb` r `:` $LS(l)$ `if` c is generated.

The syntax of the `descCheck` function is as follows.

```
op descCheck : Module -> MembAxSet .
```

The Integers Fail Confluence and Descent

Given the module `NAT` in Section 2.3, let us consider now the following specification for integers to illustrate the way in which the descent check is accomplished.

```
fmod INT is
   protecting NAT .
   sorts Int Neg .
   subsorts Zero < Neg < Int .
   subsort Nat < Int .
   ops (s_) (p_) : Int -> Int .  op p_ : Neg -> Neg .
   ops (_+_) (_*_) : Int Int -> Int [comm] .
   op -_ : Int -> Int .          op square : Int -> Nat .
   vars N M : Nat .  vars I J : Int .  vars P Q : Neg .
   eq s p I = I .                eq p s I = I .
   eq I + 0 = I .                eq I + s N = s (I + N) .
   eq I + p P = p (I + P) .      eq I * 0 = 0 .
```

```
    eq I * s N = (I * N) + I .      eq I * p P = (I * P) + - I .
    eq - 0 = 0 .                    eq - s N = p - N .
    eq - p P = s - P .              eq square(I) = I * I .
endfm
```

This specification is perfectly reasonable: it is ground-confluent, and its initial model is the ring of the integers with the usual operations, including also the square function. However, it is not confluent and if fails the descent property.

To check descent, all the instances specializing the variables to smaller sorts are generated for each equation. Let us consider, for example, the instances for the equation `square(I) = I * I`. The specializations for which descent is not satisfied are: `square(I@Neg) = I@Neg * I@Neg`, with `I@Neg` a variable of sort `Neg`, and `square(I) = I * I`, and the corresponding proof obligations generated are `I@Neg * I@Neg : Nat` and `I * I : Nat`. It is easy to see that the second one is more general than the first. Therefore, only the second one is included in the output of the tool, together with three critical pairs.

```
Checking solution :
  var I : Int . var P : Neg . var P@ : Neg .
  var N : Nat . var N@ : Nat .
  cp s - P + (- P@ + (P * P@)) = s - P@ + (- P + (P * P@)) .
  cp p - N + (P@ + (N * P@)) = p (P@ + (- N + (N * P@))) .
  cp s ( N + (N@ + (N * N@))) = s (N@ + (N + (N * N@))) .
  mb I * I : Nat .
```

Regarding the critical pairs, the first one comes from the overlapping of the equation `I * s N = (I * N) + I` with a renamed copy of itself at the top. The same critical pair is also generated in the overlapping of the equation `s N * M = M + (N * M)` with a renamed copy of itself at the top. Nevertheless, only one of them is taken; the other one is eliminated in the last step of the process. The second critical pair comes from the overlapping at the top of the equations `I * s N = (I * N) + I` and `I * p Q = (I * Q) + - I`. Finally, the last of these critical pairs comes from the overlapping of the equation `I * p Q = (I * Q) + - I` with a renamed copy of itself at the top.

The Church-Rosser Checker

In the present section we discuss the function `checking`. This function checks the Church-Rosser property of an order-sorted equational specification satisfying the conditions mentioned in the previous sections. The checking for confluence and for descent is respectively carried out by the functions `conflCheck` and `descCheck` presented above.

The output of the `checking` function is given as a 3-tuple of sort `CheckSol` consisting of a set of critical pairs, a set of membership assertions (corresponding to descent proof obligations) and the set of variables appearing in them.

As already mentioned, the modules passed to the functions `conflCheck` and `descCheck` are previously *prepared* in order to get the variables needed in the respective processes. The function `prepMod` adds to the module the variables needed in the checking. We also

define functions varDeclSet to collect the set of variables in a set of critical pairs and in a set of membership assertions.

```
op checkSol : VarDeclSet CritPairSet MembAxSet -> CheckSol .
op prepMod : Module -> Module .
op varDeclSet : Module CritPairSet -> VarDeclSet .
op varDeclSet : Module MembAxSet -> VarDeclSet .
```

Finally, the definition of the checking function is as follows.

```
op checking : Module -> CheckSol .
eq checking(M)
  = checkSol((varDeclSet(prepMod(M), conflCheck(prepMod(M)))
              varDeclSet(prepMod(M), descCheck(prepMod(M)))),
      conflCheck(prepMod(M)), descCheck(prepMod(M))) .
```

4.3. How to Use the Church-Rosser Checker

This section illustrates with examples the use of the Church-Rosser checker tool, and suggests some methods that—using the feedback provided by the tool—can help the user establish that his/her specification is ground-Church-Rosser. For additional examples see [18.10].

We assume a context of use very different from the usual context for completing an equational theory. The starting point for completing a theory, say the theory of groups, is an equational theory that is *not* Church-Rosser. A Knuth-Bendix-like completion process then attempts to make it so by *automatically adding* new oriented equations. In our case, however, we assume that the user has developed an *executable specification* of his/her intended system with an initial algebra semantics, and that this specification has already been *tested* with examples, so that the user is in fact confident that the specification is *ground-Church-Rosser*, and wants only to check this property with the tool.

Of course, the tool can only guarantee success when the user's specification is unconditional and Church-Rosser, and not just ground-Church-Rosser. That is, not generating any proof obligations is only a *sufficient* condition. But in many cases of interest the specification will typically be *ground* Church-Rosser, but not Church-Rosser, so that a collection of critical pairs and of membership assertions will be returned by the tool as proof obligations.

An important methodological question is what to do, or not do, with these proof obligations. As the examples that we discuss illustrate, what should *not* be done is to let an automatic completion process add new equations to the user's specification in a mindless way. In some cases this is even impossible, because the critical pair in question cannot be oriented. In many cases it will certainly lead to a nonterminating process. In any case, it will modify the user's specification in ways that can make it difficult for the user to recognize the final result, if any, as intuitively equivalent to the original specification.

The feedback of the tool should instead be used as a guide for *careful thought* about one's specification. As several of the examples studied indicate, by analyzing the critical pairs returned, the user can understand why they could not be joined. It may be a mistake that must be corrected. More often, however, it is not a matter of a mistake, but of an

axiom that is either *too general*—so that its very generality makes joining the critical pair impossible, because no more equations can apply to it—or *amenable to an equivalent formulation* that is unproblematic—for example, by reordering the parentheses for an operator that is ground-associative—or both. In any case, it is the user himself/herself who must study where the problem comes from, and how to fix it by modifying the specification. Interaction with the tool then provides a way of modifying the original specification and ascertaining whether the new version passes the test or is a good step towards that goal.

Since the user's specification has an *initial* algebra semantics and the property of interest is checking that it is *ground* Church-Rosser, the proof obligations returned by the tool are *inductive* proof obligations. Therefore, after having introduced some modifications that may already eliminate some of the critical pairs and membership assertions generated by the tool, the user may be left with proof obligations for which the best approach is not any further modification of the specification, but, instead, an inductive proof.

Inductive proof of the joinability of critical pairs is a thornier issue, for which we lack at present good methods. The problem is that, if we have a critical pair $\mathtt{cp}(t,\ t')$ generated by the tool for a module M, proving inductively $M \vdash_{\mathsf{ind}} t = t'$ is *not* sufficient for ensuring joinability. This is because such a proof can add new equations as lemmas that may destroy the Church-Rosser property, so that joinability with those new equations does not guarantee joinability with the original equations. What must instead be proved inductively is $M \vdash_{\mathsf{ind}} t \downarrow t'$ but this requires new inductive methods that, as far as we know, have not yet been developed.

A related unresolved methodological issue is what to do with *conditional* critical pairs or membership assertions whose conditions are *unsatisfiable*. For example, the condition may be of the form `tt = ff`, where `tt` and `ff` are constants in a user-defined Boolean sort. If we already *knew* that the specification was Church-Rosser, we could conclude from the fact that `tt` and `ff` are irreducible that they are different, so that the condition cannot be satisfied. But this is precisely what we need to prove. It is quite possible that a modular/hierarchical approach could be used, in conjunction with new inductive proof methods, to establish the unsatisfiability of such conditions and then discard the corresponding proof obligations. But such an approach has still to be developed.

Making the Natural Numbers Specification Confluent

In Section 4.2 we studied the confluence of a simple specification for the natural numbers. Let us reconsider it and analyze in more detail the checker's output, and how we can make the specification confluent. If we manage to make it confluent, since no membership assertions are generated by the tool, this would show that the specification is Church-Rosser and, a fortiori, ground-Church-Rosser.

The user of the Church-Rosser checker tool can in some cases succeed by orienting a critical pair, adding it to the specification, checking termination, and resubmitting the modified specification to the checker. However, in this case it is clear that this critical pair cannot be oriented, since, if we were to orient it, its addition would turn the specification into a nonterminating one. The way to handle this problem in our approach consists in studying the specification, and, in particular, the equations generating the critical pairs, and trying to find a "smart" solution modifying the specification in a way as intuitive and

as clear as possible, since it is the user who decides the modifications to apply. The key is then to give to the user the information needed to allow him/her to proceed as he/she thinks best.

As seen in Section 4.2, only one critical pair is generated for the specification of the natural numbers presented there. We saw how this critical pair comes from the overlap of the equation `s N * M = M + (N * M)` with a renamed copy of itself at the top. This indicates that a term of the form `s N * s M` can be rewritten to both `s (N + (M + (N * M)))` and `s (M + (N + (N * M)))`. Note that these terms cannot be further reduced, but their ground instances can be reduced: that is, the tool's output does not contradict the specification's ground confluence.

Looking to the unjoinable terms in the picture above, we can think of writing such equation as, for example, `s N * s M = s ((N + M) + (N * M))`. Replacing the original equation by this one in the input to the Church-Rosser checker, the tool does not return any critical pair. This means that, once termination has been checked, we are guaranteed that the modified specification is confluent.

Since we do not have any proof obligation for descent we can conclude that the specification is Church-Rosser.

Proving the Integers Specification Ground-Church-Rosser

Let us now apply our technique to the specification for the integers presented in Section 4.2 in order to eliminate the critical pairs discussed there, so as to make the specification confluent.

The first thing we should do is to apply the results in a modular way, that is, we should apply to the specifications the changes that have been applied on the specifications being imported. Therefore, we should incorporate the changes making the specification of the natural numbers confluent in the previous section. However, this does not necessarily mean that we are going to get fewer critical pairs or membership assertions. Notice that in the analysis of the critical pairs we saw how the first critical pair was generated from two different overlappings, one of them between one of the equations in `NAT` with itself, which was eliminated, and another one from one equation in `INT` also with a renamed copy of itself. Thus, in this example we get the same result from the tool.

Observing the considerations of Section 4.2, we realize that these critical pairs are generated for exactly the same reasons by which the critical pair for `NAT` was generated. Therefore, we can make similar modifications to `INT`. Specifically, let us replace the equations `I * s N = (I * N) + I` and `I * p P = (I * P) + - I` by the following three ones:

```
eq s N * s M = s ((N + M) + (N * M)) .
eq p P * s N = s ((P + - N) + (P * N)) .
eq p P * p Q = s ((- P + - Q) + (P * Q)) .
```

Calling the checker with these equations in the input module instead of the previous ones no critical pair is given in the output.

Regarding descent, we need to prove inductively the membership assertion given. That is, we have to treat it as the proof obligation that has to be satisfied in order to be

able to assert that the specification is ground-decreasing. In this case, we have to prove `INT` $\vdash_{\text{ind}}$ `(∀I) I * I: Nat`. This can be done using the theorem prover presented in Section 3, as was shown in [10]. Therefore, we have successfully transformed our original `INT` specification into one that is confluent and ground-descending, and therefore ground-Church-Rosser.

5. Concluding Remarks

Our experience in building and using the two tools that we have described has been very encouraging. There are several natural extensions of these tools that we have already implemented and others that we plan to pursue in the near future (check Clavel's webpage, `http://sophia.unav.es/~clavel`, for the most recent version of the `ITP`). One promising direction for extensions of the `ITP` is supporting a mixture of the current explicit induction methods and the automated induction techniques for membership equational logic proposed in [3].

Similarly, the Church-Rosser checker should be extended in several directions. First, the set of equational attributes used to rewrite modulo should be extended to contain other axioms besides commutativity. Furthermore, using the results in [3], the tool should be generalized so as to deal with membership equational logic specifications and not just order-sorted specifications. Completion should also be added as an additional facility, perhaps integrating CiME as an auxiliary termination tool. Finally, this tool should also be extended from equational logic to rewriting logic, to check *coherence* [40] of rewrite theories, or to complete such theories so that they become coherent.

More broadly, the present experience suggests that the reflective approach that we have taken in building the present tools is a promising general methodology to build many other theorem proving and formal analysis tools based on formal systems for different logics. Besides the two tools described in this paper, there is in fact a substantial body of experimental evidence supporting the feasibility and convenience of using Maude as a *formal metatool* to generate formal tools from their declarative specifications. The paper [11] discusses several other such formal tool generator applications developed by different authors, including: (1) formal translation tools from HOL to Nuprl, from linear logic to rewriting logic [30,7], from the Wright architectural description language to CSP and from CSP to rewriting logic, and from pure (higher-order) type systems to rewriting logic [38]; (2) a proof assistant for the open calculus of constructions (an equational extension of the calculus of constructions) [38], and (3) tools for formal specification languages such as the Full Maude tool for Maude [18,19], the definition in Maude by Millen and Denker of their Common Authentification Specification Language (CAPSL) and its translation into the CIL intermediate language [16], and the execution environment for Tile Logic in rewriting logic [4,5].

An important added benefit of building a formal environment of tools this way is that, since each tool is a theory in rewriting logic, they are much easier to interoperate by just combining their corresponding rewrite theories. We have for example seen the usefulness of using the inductive theorem prover to prove proof obligations generated by the Church-Rosser checker, and have explained how both tools are integrated with the Full Maude environment. In general, using the reflective techniques and the flexible logical framework

capabilities of rewriting logic, we hope to make good advances towards the goal of *formal interoperability* [36], that is, the capacity to move in a mathematically rigorous way across different formalizations, and to use in a rigorously integrated way the different tools supporting such formalizations.

Acknowledgments

We have benefited very much from discussions and positive suggestions from Patrick Lincoln and Natarajan Shankar at SRI, and from those of our fellow members in the Cafe Project, particularly from Kokichi Futatsugi, Ataru Nakagawa, Toshimi Sawada, Masaki Ishiguro, and Masayuki Fujita. Thanks to their efforts, earlier versions of our tools were successfully integrated within the Cafe environment. The present paper is based on extensions of both tools based on the current metalevel of Maude [9] and having convenient user interfaces defined within Maude and a number of additional features.

The theoretical foundations of, and automated deduction techniques for, the tools have benefited very much from our ongoing collaboration with Adel Bouhoula and Jean-Pierre Jouannaud on theorem proving techniques for membership equational logic [3], and from very fruitful discussions with them during the Cafe project.

The explicit induction approach adopted for the inductive theorem prover extends in a reflective way a similar explicit induction approach to equational theorem proving in OBJ proposed by Joseph Goguen [22,23]. We have benefited considerably from Joseph Goguen's advice and from the OBJ experience in the design of the inductive theorem prover.

REFERENCES

1. F. Baader and T. Nipkow. *Term Rewriting and* all that. Cambridge University Press, 1999.
2. J. Bergstra and J. Tucker. Characterization of computable data types by means of a finite equational specification method. In J. W. de Bakker and J. van Leeuwen, eds., *Seventh Colloquium on Automata, Languages and Programming*, vol. 81 of *Lecture Notes in Computer Science*, pp. 76–90. Springer-Verlag, 1980.
3. A. Bouhoula, J.-P. Jouannaud, and J. Meseguer. Specification and proof in membership equational logic. *Theoretical Computer Science*, 236(1):35–132, 2000.
4. R. Bruni, J. Meseguer, and U. Montanari. Process and term tile logic. Technical Report SRI-CSL-98-06, SRI International, July 1998.
5. R. Bruni, J. Meseguer, and U. Montanari. Internal strategies in a rewriting implementation of tile systems. In Kirchner and Kirchner [28].
6. CafeOBJ-Project. *Procs. of the CafeOBJ Symposium'98*. 1998.
7. M. Clavel. *Reflection in general logics and in rewriting logic with applications to the Maude language*. PhD thesis, U. of Navarre, 1998.
8. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, and J. Meseguer. Metalevel computation in Maude. In Kirchner and Kirchner [28].
9. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. Quesada. Maude: Specification and programming in rewriting logic. Manuscript, SRI International, January 1999. Available at `http://maude.csl.sri.com`.

10. M. Clavel, F. Durán, S. Eker, and J. Meseguer. Design and implementation of the Cafe prover and the Church-Rosser checker tools. Manuscript, SRI International, February 1998.
11. M. Clavel, F. Durán, S. Eker, J. Meseguer, and M.-O. Stehr. Maude as a formal meta-tool. In J. Wing, J. Woodcock, and J. Davies, eds., *Formal Methods*, vol. 1709 of *Lecture Notes in Computer Science*, pp. 1684–1704. Springer-Verlag, 1999.
12. M. Clavel, S. Eker, P. Lincoln, and J. Meseguer. Principles of Maude. In Meseguer [34].
13. M. Clavel and J. Meseguer. Axiomatizing reflective logics and languages. In G. Kiczales, ed., *Procs. of Reflection*, 1996.
14. M. Clavel and J. Meseguer. Reflection and strategies in rewriting logic. In Meseguer [34].
15. E. Contejean and C. Marché. The CiME system: tutorial and user's manual. Manuscript, Université Paris-Sud, Centre d'Orsay.
16. G. Denker and J. Millen. CAPSL intermediate language. In N. Heintze and E. Clarke, eds., *Workshop on Formal Methods and Security Protocols*, 1999.
17. R. Diaconescu and K. Futatsugi. *CafeOBJ Report*. AMAST Series. World Scientific, 1998.
18. F. Durán. *A Reflective Module Algebra with Applications to the Maude Language*. PhD thesis, U. de Málaga, 1999.
19. F. Durán and J. Meseguer. An extensible module algebra for Maude. In Kirchner and Kirchner [28].
20. K. Futatsugi and T. Sawada. Cafe as an extensible specification environment. In *Procs. of Kunming International CASE Symposium*, 1994.
21. I. Gnaedig, C. Kirchner, and H. Kirchner. Equational completion in order-sorted algebras. *Theoretical Computer Science*, 72:169–202, 1990.
22. J. Goguen. OBJ as a theorem prover with application to hardware verification. In P. Subramanyam and G. Birtwistle, eds., *Current Trends in Hardware Verification and Automated Theorem Proving*, pp. 218–267. Springer-Verlag, 1989.
23. J. Goguen. Theorem proving and algebra. Manuscript, August 1997.
24. J. Goguen and J. Meseguer. Order-sorted algebra I: Equational deduction for multiple inheritance, overloading, exceptions and partial operations. *Theoretical Computer Science*, 105:217–273, 1992.
25. J. Goguen, T. Winkler, J. Meseguer, K. Futatsugi, and J.-P. Jouannaud. Introducing OBJ. Technical Report SRI-CSL-92-03, SRI International, 1992.
26. J. V. Guttag and J. J. Horning. *Larch: Languages and Tools for Formal Specification*. Springer-Verlag, 1993.
27. C. Kirchner. Order-sorted equational unification. *Procs. of 5th Intl. Conf. on Logic Programming*, 1988.
28. C. Kirchner and H. Kirchner, eds. *Procs. 2nd Intl. Workshop on Rewriting Logic and its Applications*, vol. 15 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1998. Available at `http://www.elsevier.nl/locate/entcs/volume15.html`.
29. C. Kirchner, H. Kirchner, and J. Meseguer. Operational semantics of OBJ3. In T. Lepistö and A. Salomaa, eds., *Procs. of 15th Intl. Coll. on Automata, Languages and Programming*, vol. 317 of *Lecture Notes in Computer Science*, pp. 287–301.

Springer, 1988.

30. N. Martí-Oliet and J. Meseguer. Rewriting logic as a logical and semantic framework. Technical Report SRI-CSL-93-05, SRI International, August 1993. To appear in D.M. Gabbay, ed., *Handbook of Philosophical Logic*, Kluwer Academic Publishers.
31. N. Martí-Oliet and J. Meseguer. Rewriting logic as a logical and semantic framework. In Meseguer [34].
32. J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.
33. J. Meseguer. A logical theory of concurrent objects and its realization in the Maude language. In G. Agha, P. Wegner, and A. Yonezawa, eds., *Research Directions in Concurrent Object-Oriented Programming*, pp. 314–390. MIT Press, 1993.
34. J. Meseguer, editor. *Procs. of the 1st International Workshop on Rewriting Logic and its Applications*, vol. 4 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1996. Available at `http://www.elsevier.nl/locate/entcs/volume4.html`.
35. J. Meseguer. Rewriting logic as a semantic framework for concurrency: a progress report. In U. Montanari and V. Sassone, eds., *Proc. of 7th Intl. Conf. on Concurrency Theory*, vol. 1119 of *Lecture Notes in Computer Science*, pp. 331–372. Springer-Verlag, 1996.
36. J. Meseguer. Formal interoperability. In *Procs. of the 1998 Conf. on Mathematics in Artificial Intelligence*, 1998.
37. J. Meseguer. Membership algebra as a semantic framework for equational specification. In F. Parisi-Presicce, ed., *Recent Trends in Algebraic Development Techniques*, vol. 1376 of *Lecture Notes in Computer Science*, pp. 18–61. Springer-Verlag, 1998.
38. J. Meseguer and M.-O. Stehr. Pure type systems in rewriting logic — meta-logical and meta-operational views. Submitted for publication.
39. J. Meseguer and T. Winkler. Parallel programming in Maude. In J.-P. Banâtre and D. L. Mètayer, eds., *Research Directions in High-level Parallel Programming Languages*, vol. 574 of *Lecture Notes in Computer Science*, pp. 253–293. Springer-Verlag, 1992.
40. P. Viry. Rewriting: An effective model of concurrency. In C. Halatsis et al., eds., *Proc. of 6th Int. Conf. on Parallel Architectures and Languages Europe*, vol. 817 of *Lecture Notes in Computer Science*, pp. 648–660. Springer-Verlag, 1994.

This Page Intentionally Left Blank

Chapter 2

CafeOBJ Jewels

Răzvan Diaconescu[a] and Kokichi Futatsugi and Shusaku Iida[b]

[a]Institute of Mathematics "Simion Stoilow", Romania

[b]Japan Advanced Institute for Science and Technology

This paper gives an overview of the main features and methodologies of CafeOBJ by means of a collection of elegant examples. Therefore, this paper may also serve as a tutorial introduction to CafeOBJ. We hope that besides the strength of CafeOBJ, the reader will also appreciate the beauty of this language.

1. Introduction

1.1. Overview of CafeOBJ

CafeOBJ is an *executable* industrial strength algebraic specification language which is a modern successor of OBJ and incorporating several new algebraic specification paradigms. Its definition is given in [8]. CafeOBJ is intended to be mainly used for system specification, formal verification of specifications, rapid prototyping, programming, etc. Here is a brief overview of its most important features.

Equational Specification and Programming.

This is inherited from OBJ [21,12] and constitutes the basis of the language, the other features being somehow built on top of it. As with OBJ, CafeOBJ is *executable* (by term rewriting), which gives an elegant declarative way of functional programming, often referred as *algebraic programming.*[1] As with OBJ, CafeOBJ also permits equational specification modulo several equational theories such as associativity, commutativity, identity, idempotence, and combinations between all these. This feature is reflected at the execution level by term rewriting *modulo* such equational theories.

Behavioural Specification.

Behavioural specification [16,17,9] provides a novel generalisation of ordinary algebraic specification. Behavioural specification characterises how objects (and systems) *behave*, not how they are implemented. This new form of abstraction can be very powerful in the specification and verification of software systems since it naturally embeds other useful paradigms such as concurrency, object-orientation, constraints, nondeterminism, etc. (see [17] for details). Behavioural abstraction is achieved by using specification with hidden

[1]Please notice that although this paradigm may be used as programming, this aspect is still secondary to its specification side.

sorts and a behavioural concept of satisfaction based on the idea of indistinguishability of states that are observationally the same, which also generalises process algebra and transition systems (see [17]).

CafeOBJ directly supports behavioural specification and its proof theory through special language constructs, such as

- hidden sorts (for states of systems),
- behavioural operations (for direct "actions" and "observations" on states of systems),
- behavioural coherence declarations for (non-behavioural) operations (which may be either derived (indirect) "observations" or "constructors" on states of systems), and
- behavioural axioms (stating behavioural satisfaction).

The advanced coinduction proof method receives support in CafeOBJ via a default (candidate) coinduction relation (denoted =*=). In CafeOBJ, coinduction can be used either in the classical HSA sense [17] for proving behavioural equivalence of states of objects, or for proving behavioural transitions (which appear when applying behavioural abstraction to RWL).[2]

Besides language constructs, CafeOBJ supports behavioural specification and verification by several methodologies.[3] CafeOBJ currently highlights a methodology for concurrent object composition which features high reusability not only of specification code but also of verifications [8,23]. Behavioural specification in CafeOBJ may also be effectively used as an object-oriented (state-oriented) alternative for traditional data-oriented specifications. Experiments seem to indicate that an object-oriented style of specification even of basic data types (such as sets, lists, etc.) may lead to higher simplicity of code and drastic simplification of verification process [8].

Behavioural specification is reflected at the execution level by the concept of *behavioural rewriting* [8,9] which refines ordinary rewriting with a condition ensuring the correctness of the use of behavioural equations in proving strict equalities.

Rewriting Logic Specification.

Rewriting logic specification in CafeOBJ is based on a simplified version of Meseguer's *rewriting logic* [25] specification framework for concurrent systems which gives a non-trivial extension of traditional algebraic specification towards concurrency. RWL incorporates many different models of concurrency in a natural, simple, and elegant way, thus giving CafeOBJ a wide range of applications. Unlike Maude [3], the current CafeOBJ design does not fully support *labelled* RWL which permits full reasoning about multiple transitions between states (or system configurations), but provides proof support for reasoning about the *existence* of transitions between states (or configurations) of concurrent systems via a

[2]However, until the time this paper was written, the latter has not been yet explored sufficiently, especially practically.

[3]This is still an open research topic, the current methodologies may be developed further and new methodologies may be added in the future.

built-in predicate (denoted ==>) with dynamic definition encoding both the proof theory of RWL and the user defined transitions (rules) into equational logic.

From a methodological perspective, CafeOBJ develops the use of RWL transitions for specifying and verifying the properties of *declarative encoding of algorithms* (see [8]) as well as for specifying and verifying transition systems.

Module System.

The principles of the CafeOBJ module system are inherited from OBJ which builds on ideas first realized in the language Clear [2], most notably institutions [14,11]. CafeOBJ module system features

- several kinds of imports,
- sharing for multiple imports,
- parameterised programming allowing
  - multiple parameters,
  - views for parameter instantiation,
  - integration of CafeOBJ specifications with executable code in a lower level language
- module expressions.

However, the theory supporting the CafeOBJ module system represents an updating of the original Clear/OBJ concepts to the more sophisticated situation of multi-paradigm systems involving theory morphisms across institution embeddings [5], and the concrete design of the language revise the OBJ view on importation modes and parameters [8].

Type System and Partiality.

CafeOBJ has a type system that allows subtypes based on *order sorted algebra* (abbreviated **OSA**) [20,15]. This provides a mathematically rigorous form of runtime type checking and error handling, giving CafeOBJ a syntactic flexibility comparable to that of untyped languages, while preserving all the advantages of strong typing.

We decided to keep the concrete order sortedness formalism open at least at the level of the language definition. Instead we formulate some basic simple conditions which any concrete CafeOBJ order sorted formalism should obey. These conditions come close to Meseguer's OSA^R [26] which is a revised version of other versions of order sortedness existing in the literature, most notably Goguen's OSA [15].

CafeOBJ does not directly do partial operations but rather handles them by using error sorts and a sort membership predicate in the style of *membership equational logic* (abbreviated **MEL**) [26]. The semantics of specifications with partial operations is given by MEL.

Logical semantics.

CafeOBJ is a declarative language with firm mathematical and logical foundations in the same way as other OBJ-family languages (OBJ, Eqlog [18,4], FOOPS [19], Maude [25]) are. The reference paper for the CafeOBJ mathematical foundations is [6], while the book [8] gives a somehow less mathematical easy-to-read (including many examples) presentation of the semantics of CafeOBJ. In this section we give a very brief overview of the CafeOBJ logical and mathematical foundations, for a full understanding of this aspect of CafeOBJ the reader is referred to [6] and [8].

The mathematical semantics of CafeOBJ is based on state-of-the-art algebraic specification concepts and results, and is strongly based on category theory and the theory of institutions [14,5,11]. The following are the principles governing the logical and mathematical foundations of CafeOBJ:

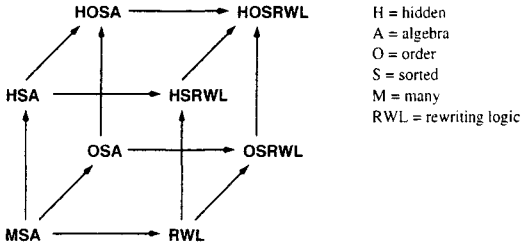
P1. there is an underlying logic[4] in which all basic constructs and features of the language can be rigorously explained.

P2. provide an integrated, cohesive, and unitary approach to the semantics of specification in-the-small and in-the-large.

P3. develop all ingredients (concepts, results, etc.) at the highest appropriate level of abstraction.

The CafeOBJ cube.

CafeOBJ is a multi-paradigm language. Each of the main paradigms implemented in CafeOBJ is rigorously based on some underlying logic; the paradigms resulting from various combinations are based on the combination of logics. The structure of these logics is shown by the following **CafeOBJ cube** , where the arrows mean embedding between the logics, which correspond to institution embeddings (i.e., a strong form of institution morphisms of [14,11]) (the orientation of arrows correspond to embedding "less complex" into "more complex" logics).



[4]Here "logic" should be understood in the modern relativistic sense of "institution" which provides a mathematical definition for a logic (see [14]) rather than in the more classical sense.

The mathematical structure represented by this cube is that of a *lattice of institution embeddings* [5,6]. By employing other logical-based paradigms the CafeOBJ cube may be thought as a hyper-cube (see [6,8] for details). It is important to understand that the CafeOBJ logical foundations are based on the CafeOBJ cube rather than on its flattening represented by HOSRWL.[5]

The design of CafeOBJ lead to several important developments in algebraic specification theory. One of them is the concept of *extra theory morphism* [5], which is a concept of theory morphism across institution embeddings, generalising the ordinary (intra) theory morphisms to the multi-paradigm situation. Another important theoretical development is constituted by the formalism underlying behavioural specification in CafeOBJ which is a non-trivial extension of classical hidden algebra [17] in several directions, most notably permitting operations with several hidden arguments via the crucial *coherence* property. This extension is called "coherent hidden algebra" (abbreviated **CHA**) in [9] and comes very close to the "observational logic" of Bidoit and Hennicker [22]. The details of the "coherent hidden algebra" institution can be found in [6].

1.2. CafeOBJ Basic References

The definition of the language is reported in the book [8], the logical semantics of the language first appears in [7] and is further refined in [6]. The underlying mathematics on institutions supporting the logical semantics appears in [5] and the mathematical foundations for the behavioural specification paradigm in CafeOBJ can be found in [9]. Finally, [10] gives a recent overview of the behavioural specification methodologies in CafeOBJ.

The CafeOBJ home page language at http://www.ldl.jaist.ac.jp/cafeobj provides pointers to various aspects of the CafeOBJ research, including both theoretical works, methodological works, the system, manuals, examples, and various applications.

1.3. Brief Overview of the Examples

The first example presented is a very compact specification of a sorting algorithm. The main point of this example is to illustrate how CafeOBJ can be used both for executing algorithms and for reasoning about the algorithm itself. Besides presenting basic data type specification and RWL specification in CafeOBJ, we also discuss special CafeOBJ features such as specification/programming modulo equational attributes, use of subsorts, use of parameterised modules and module expressions, and proof techniques both for equational and RWL specifications.

The second part of the paper is devoted to CafeOBJ handling of nondeterminism. Among many possible ways to deal with nondeterminism, we present and compare two different methods, one based on RWL, and the other based on behavioural specification. In both examples we address the same problem, i.e., the specification of nondeterministic natural numbers. We also illustrate how to prove properties about nondeterminism in both approaches, and discuss the relative strength of each of them with respect to

[5]The technical reason for this is that model reducts across some of the edges (i.e., the left-to-right ones) of the CafeOBJ cube involve both an ordinary model reduct and a non-trivial reduct along the corresponding institution embedding, see [6,5,8] for details.

doing proofs. The behavioural specification example also serves as an introduction to behavioural specification and verification in CafeOBJ, featuring some of the support which CafeOBJ provides for behavioural specification and verification (including hidden sorts, behavioural operations. behavioural coherent operations, default coinduction relation, etc.).

The third part of the paper goes deeper into the behavioural specification side of CafeOBJ by doing "behavioural" lists and sets in an object-oriented style which is contrasted to the traditional data-oriented style. We illustrate a behavioural specification style in CafeOBJ based on extensive use of behavioural coherence, which further simplifies proofs of properties for these specifications.

The final section is dedicated to the current CafeOBJ methodology for composing objects concurrently by using the so-called "projection operations". We develop the example of a dynamic bank accounts system, and prove some correctness properties stated as behavioural properties at the level of the whole system. These proofs also features a method for automatic generation of case analysis by the system by a meta-level encoding in RWL; this is especially useful when having to deal with a large number of cases. The final part of this section deals with synchronisation between the components via a counter-with-switch example.

2. Sorting Strings

Most of the running time of todays computers may still be devoted to sorting and searching. Sorting programs are used everywhere, and it is not surprising that much research has been done for improving the sorting algorithms. For example, Knuth's "programming bible" devotes a whole volume only to sorting and searching [24].

So, let us see how we can do sorting in CafeOBJ. Since CafeOBJ is both a programming and a specification language, we can do more that just sort things, we can also *reason about the sorting algorithm* itself. In this section we deliberately stay away of efficiency problems (which in the case of sorting is a main theme) since we want to concentrate on the programming, specification, and formal verification aspects of CafeOBJ.

2.1. Specifying Strings

In order to proceed with sorting, we need a data type to be sorted. This data type need not be a concrete one, it can be a *generic* data type. So, let us consider strings over partial orders.

We can start by specifying just generic strings, and then to instantiate those to strings over partial orders.

```
mod! STRG(X :: TRIV) {  [ Elt  <  Strg ]
 op nil : -> Strg
 op __ : Strg Strg -> Strg {assoc idr: nil}
}
```

Here we just say that generic strings take elements from any set. "Any set" is specified by the parameter TRIV:

```
mod* TRIV { [ Elt ] }
```

which contains only one sort, Elt and is provided as a system built-in module. Because

this is a loose semantics declaration (**mod***), the models of this specification consists of all (plain) sets.

Then, the subsort declaration [Elt < Strg] says that each element of the given set should be regarded as a string (with only one element). A string is built from atomic elements by using the concatenation operation __ (notice the use of a *mixfix* notation). There is also an empty string *nil*. The concatenation operation has the usual properties: associativity (which means there is no need for bracketing strings), and *nil* is an identity for the concatenation operation. Notice that both associativity and identity equations are actually specified as operation attributes rather than ordinary equations. This is a rather subtle feature of CafeOBJ, the computation being done *modulo* these equations rather than involving them directly in the computation. These sentences (axioms) correspond exactly to the algebraic structure of monoids. The *initial denotation* declaration **mod!** means that from all monoids generated by a fixed interpretation of Elt, we consider only the strings model as the denotation for this specification.

Now we can obtain strings of anything we want just by instantiating the generic set of elements to any chosen data type. In our case, we need a data type which has some ordering, and the most generic one is just the theory of partially ordered sets:

```
mod* POSET { [ Elt ]
  pred _<=_ : Elt Elt
  vars E1 E2 E3 : Elt
  eq E1 <= E1 = true .
  cq E1 = E2 if (E1 <= E2) and (E2 <= E1) .
  cq (E1 <= E3) = true if (E1 <= E2) and (E2 <= E3) .
}
```

Here the partial ordering relation is specified as the binary predicate _<=_, which in CafeOBJ is the same as a Bool-valued operation:

```
op _<=_ : Elt Elt -> Bool
```

The sort Bool, as well as the constant true and the operation _and_ are part of built-in Boolean data type BOOL which is available (by default importation) in any module (unless otherwise explicitly specified). The condition of the conditional equations (cq or ceq) is just a Bool-sorted term. The denotation of POSET consists of all partial orders.

The instantiation is done by using a *view* which is a signature morphism between the signature of the parameter (TRIV) and the signature of the value (POSET):

```
view poset from TRIV to POSET { sort Elt -> Elt }
```

Now, the strings over partial orders can be obtained just as

```
STRG(X <= poset)
```

but since the view *poset* is a default view, we can use the short-hand notation:

```
STRG(POSET)
```

Notice that in CafeOBJ, STRG(POSET) can still be regarded as a parameterized module, with *X.poset* :: POSET as parameter.

2.2. Specifying a Sorting Algorithm

We now specify a generic crude version of the bubble sort algorithm:

```
mod! SORTING-STRG (Y :: POSET) {
  protecting(STRG(Y))
```

```
  ctrans E:Elt E':Elt  =>  E' E  if  (E' <= E) and (E =/= E') .
}
```

The import of the strings is protecting, which means that there is no collapse of strings and no new elements of sort Strg, so the sorting algorithm does not change the data type of strings. Also, as you may have already guessed, the built-in Bool-valued predicate =/= means non-equality.

Notice that the sorting algorithm is specified very compactly (in just one line!) by using a (conditional) *transition* sentence expressing the swapping of elements in a string. The meaning of CafeOBJ transitions is that of *change* rather than equality, so each application of transitions just changes a string to another string. The sorting process is just a chain of such transitions, until no transition can be applied anymore. We do not need to worry where the swappings are applied because the computation is done modulo associativity, so the system finds by itself where are the appropriate places for doing swappings.

There are two basic questions regarding this sorting process: whether it always terminates, and whether it always gives a unique value ("always" meaning "for every string over any partial order"). We will answer these two questions later, after showing how sorting can be executed in CafeOBJ.

2.3. Executing the Sorting Algorithm

This algorithm is a generic one, in order to execute it we need to instantiate it to an algorithm over a concrete data type. Let us consider the natural numbers with the usual ordering. We may use a *built-in* data type of the naturals (NAT) provided by the CafeOBJ system:

```
SORTING-STRG(Y <= view nat to NAT { sort Elt -> Nat,
                                    op _<=_ -> _<=_ })
```

or just (since this is a default view too)

```
SORTING-STRG(NAT)
```

Now let's sort a string:

```
select SORTING-STRG(NAT)
exec (4 3 5 3 1) .
```

and get

```
- execute in SORTING-STRG(NAT) : 4 3 5 3 1
1 3 3 4 5 : Strg
```

This shows the CafeOBJ capability of executing algorithms. In the following we see that we can do something more interesting, to reason about this algorithm.

2.4. Reasoning about the Sorting Algorithm

In this section we give an example of reasoning about generic algorithms in CafeOBJ. We will prove two important properties of the generic sorting algorithm: termination and confluence. The proofs are *generic* in the sense that they (and their conclusion, of course) apply to sorting strings over any partial order.

The built-in predicate ==>.

In the case of algorithms specified in RWL, the built-in CafeOBJ predicate _==>_ plays a major rôle at the testing and verification stage. This predicate evaluates to true whenever there exists a transition from the left hand side argument to the right hand side argument.

```
- reduce in SORTING-STRG(NAT) : 1 5 4 3 2 ==> 1 4 3 5 2
true : Bool
```

Notice that we use here the command reduce, which computes only with the equations, thus discarding the direct use of user-defined transitions. However the user-defined transitions are used indirectly via the dynamic definition of the predicate ==>.

Proving generic termination.

One of the properties of algorithms of great practical importance is termination. We will now give a *formal proof* in CafeOBJ for the sorting algorithm of SORTING-STRG. In order to do this we introduce a function measuring the "disorder degree" of a string and show that each application of a transition step decreases this disorder degree.

```
mod! SORTING-STRG-PROOF {
  protecting(SORTING-STRG + NAT)
  op disorder : Strg -> Nat
  op _>>_ : Elt Strg -> Nat
  vars E E' : Elt
  var S : Strg
  eq disorder(nil) = 0 .
  eq disorder(E) = 0 .
  eq disorder(E S) = disorder(S) + (E >> S) .
  eq E >> nil = 0 .
  cq E >> E' = 0 if E <= E' .
  cq E >> E' = 1 if (E' <= E) and (E =/= E') .
  cq E >> (E' S) = s(E >> S) if (E' <= E) and (E =/= E') .
  cq E >> (E' S) = (E >> S) if E <= E' .
}
```

The function _>>_ just counts how many elements smaller that the first argument (element) appear in the second argument (string) and is used for the definition of *disorder*. (Notice that *s(_)* is just the built-in successor function on the naturals.)

The fact that the sorting algorithm always terminates can be then formulated as

$$s \Longrightarrow s' \text{ implies } \mathit{disorder}(s') < \mathit{disorder}(s) \text{ (whenever } s \text{ and } s' \text{ are different)}$$

since the natural numbers are well-founded, so any strictly decreasing chain of natural numbers is finite.

In order to prove this we need two lemmas:

Lemma 2.1 $\mathit{disorder}(e'\ e\ s) < \mathit{disorder}(e\ e'\ s)$ *if* $e' < e$.

Proof:

```
open SORTING-STRG-PROOF .
  ops e e' :  -> Elt .
  op s :  -> Strg .
```

The following is the working hypothesis:

```
eq e' <= e = true .
```

and here is the conclusion:

```
- reduce in %(Y.SORTING-STRG) : disorder(e' e s) < disorder(e e' s)
true : Bool
```

and

Lemma 2.2 *disorder*$(e\ s) <$ *disorder*$(e\ s')$ *if* $(e >> s)==(e >> s')$ *and* *disorder*$(s) <$ *disorder*(s').

Proof:

```
open SORTING-STRG-PROOF .
 op e : -> Elt .
 ops s s' : -> Strg .
```

The following is the working hypothesis:

```
 eq (e >> s) = (e >> s') .
 eq disorder(s) < disorder(s') = true .
```

and this is the conclusion:

```
- reduce in %(Y.SORTING-STRG) : disorder(e s) < disorder(e s')
true : Bool
```

Notice that in the proofs of the previous lemmas we used some ad-hoc constants (i.e., e, e', s, s') playing the rôle of universally quantified variables. This kind of proof technique originates from OBJ and is justified by the so-called *Theorem of Constants* (see [13]). The command **open** gives the possibility of adding temporary declarations to a module, and its opposite command **close** cancels the temporary declarations.

Proving generic confluence

The generic confluence of the sorting algorithm means that the relation ==> is confluent whatever partial order we chose for the elements of the strings. Combined with termination, this means that we always get a unique result when running the bubble sort algorithm.

Theorem 2.1 *The algorithm SORTING-STRG is locally confluent.*

Proof: We have to assume two different swappings of elements and prove that from each of the resulting strings we can reach the same string by applying the sorting algorithm. We distinguish two cases:

1. the positions of the elements involved in the two different swappings are disjoint, and
2. the positions of the elements involved in the two different swappings overlap.

The CafeOBJ proof score for case 1. is as follows:

```
open SORTING-STRG .
 ops e e' e1 e1' : -> Elt .
```

```
  ops s s' s'' : -> Strg .
```
For the initial swappings we assume the following:
```
  eq e' <= e = true .
  eq e1' <= e1 = true .
```
and now we prove the local confluence:
```
  red (s e' e s' e1 e1' s'') ==> (s e' e s' e1' e1 s'') .
  red (s e e' s' e1' e1 s'') ==> (s e' e s' e1' e1 s'') .
  close
```
The proof score for case 2. is as follows:
```
  open SORTING-STRG .
    ops e e' e'' : -> Elt .
    ops s s' : -> Strg .
```
with the following hypothesis:
```
    eq e' <= e = true .
    eq e'' <= e' = true .
```
and the following lemma (we omit here its trivial proof score):
```
    eq e'' <= e = true .
```
and now we prove the local confluence:
```
  red (s e' e e'' s') ==> (s e'' e' e s') .
  red (s e e'' e' s') ==> (s e'' e' e s') .
  close
```

Now we may use the famous Newman's Lemma (applied to the case of the transition relation ==>) since we proved the sorting algorithm is terminating.

Theorem 2.2 (***Newman Lemma***) *A relation is confluent whenever it is terminating and locally confluent.*

For more details on this basic result the reader may wish to consult [13].

Corollary 2.1 *For the sorting algorithm SORTING-STRG, the relation ==> is confluent.*

3. Nondeterminism

Nondeterminism can be handled in CafeOBJ in several different ways corresponding to several different paradigms implemented by CafeOBJ, such as rewriting logic and behavioural specification. In this section we illustrate comparatively two different ways to treat nondeterminism by means of a simple example: the nondeterministic choice of natural numbers.

3.1. In Rewriting Logic

This is an example used in the literature [25] to illustrate the semantics and the power of rewriting logic.
```
  mod! NNAT-RWL {
    extending(NAT)
    op _|_ : Nat Nat -> Nat
    vars M N : Nat
```

```
  trans N | M => N .
  trans N | M => M .
}
```

In this example we used the built-in module NAT of the natural numbers. The nondeterministic choice is modelled via the operation _|_ and the two transitions.

Notice that the sort of natural numbers gets "nondeterministic naturals" as new elements, hence the importation mode used is extending. Transitions are responsible for the nondeterministic choice, and notice that in this example the transition relation is *not* confluent. All operations inherited from NAT are extended automatically on the nondeterministic naturals. For example, a property such as

$$3 \leq (4 \mid 4 \mid 5)$$

means that whatever choice we make from (4 | 4 | 5) the result is not going to be less than 3. The CafeOBJ proof score for this property is as follows:

```
open NNAT .
red (3 <= ( 4 | 4 | 5 )) ==> false .
```

The result of running this proof score is false, which means that there is no possible transition from *3 <= (4 | 4 | 5)* to false. Ultimately, this means that *3 <= (4 | 4 | 5) for all possible transitions*, which is exactly what we want.

3.2. In Behavioural Specification

The CHA version of this problem has a very simple specification too. It is interesting to mention that this very simple solution for the non-deterministic choice of naturals in CHA was obtained by the first author after dense interaction over InterNet with Dorel Lucanu and Joseph Goguen and is the main source for the discovery of the crucial concept of behavioural coherence in CHA.

```
mod* NNAT-HSA {
  protecting(NAT)
  *[ NNat ]*
  op [_] : Nat -> NNat
  op _|_ : NNat NNat -> NNat {coherent}
  bop _->_ : NNat Nat -> Bool
  vars S1 S2 : NNat
  vars M N : Nat
  eq [M] -> N = M == N .
  eq (S1 | S2) -> N = (S1 -> N) or (S2 -> N) .
}
```

This specification has a strong object-oriented flavour. The non-deterministic naturals are modelled as an object with one "observation" (attribute), _->_, stating the possibility of choice of a natural number from a certain state of the object. This object may be implemented in various ways, hence the *loose* denotation declaration (**mod***). The space of the states for the object is denoted by the *hidden* sort NNat (notice the slightly different notation from the declaration of ordinary "visible" data sorts).

In behavioural specification the equality relation of interest is the *behavioural equivalence* relation (denoted as $\sim$) rather than the strict equality. Two states are behaviourally

equivalent iff they evaluate to the same data for all possible *behavioural contexts*. Here by "behavioural context" we mean any chain of *behavioural operations* with result of a "visible" sort. Behavioural operations are denoted in CafeOBJ by the special keyword bop and they are subjected to a condition of monadicity meaning that they admit exactly one hidden sort in their arity (the list of sorts of the arguments). So, in NNAT-HSA there is only one bop, namely the "observation" _->_. This means that in this case the behavioural equivalence relation is pretty simple:

$$s \sim s' \text{ iff } (s \texttt{ -> } n) == (s' \texttt{ -> } n) \text{ for all naturals } n$$

Such simple behavioural equivalence (i.e., just equality under *all* "observation" evaluations) is rather common, especially for smaller objects. CafeOBJ provides syntactic and computational support for these situations via the built-in "hidden" relation =*= which is defined exactly as equality under *all* "observation" evaluations.

This example highlights another important feature of CafeOBJ, namely *behaviourally coherent operations*, i.e., ordinary operations preserving the behavioural equivalence relation. The nondeterministic choice is modelled via the (non-behavioural) operation _|_, but this operation preserves the behavioural equivalence relation. This has several consequences, like soundness of the ordinary equational deduction rules for behavioural equality, and more computation power when verifying beahvioural specifications in CafeOBJ.

In this setup, associativity, commutativity, or idempotence (which are natural expected properties of nondeterministic choice) appear as behavioural properties and they have very simple proof scores. Take commutativity, for example:

```
open NNAT-HSA .
  ops s1 s2 : -> NNat .
  op n :  -> Nat .
red (s1 | s2) -> n == (s2 | s1) -> n .
```

A property such as $3 \leq (4 \mid 4 \mid 5)$ gets again very simple proof, but at the cost of extension by hand of the predicate $\leq$ as an "observation" on NNat:

```
mod* NNAT-HSA<= {
  protecting(NNAT-HSA)
  bop _<=_ : Nat NNat -> Bool
  vars M N : Nat
  vars S1 S2 : NNat
  eq  N <= [M] = N <= M .
  eq N <= (S1 | S2) = (N <= S1) and (N <= S2) .
}
```

Then the proof is as follows:

```
open NNAT-HSA<= .
red 3 <= ([4] | [4] | [5]) .
```

Notice also the complexity of such computation is $O(n)$ while the complexity of the corresponding RWL computation is $O(n!)$ since behavioural abstraction allows cutting dramatically the transition space. On the other hand, the relative disadvantage of having to extend by hand the built-in predicates (operations) from Nat to NNat can be overcome by the use of parameterization since the equations corresponding to these extensions are essentially the same.

4. Behavioural Sets and Lists

In this section we discuss the *basic behavioural specification* paradigm which proposes an object-oriented algebraic specification style as opposed to the more classical data-oriented one. For this, we use the examples of sets and lists, and show that lists are a (behavioural) refinement of sets, hence sets can be implemented as lists (which is a usual implementation technique for sets, for example this is how sets are implemented in LISP).

4.1. Behavioural Lists

Lists constitute one of the most intensively used data types. Behavioural specification provides a strongly contrasting specification of lists with respect to the traditional data-oriented ones based on initial algebra semantics. As we will see, behavioural specification has the great advantage of *simplicity* (mainly due to the *loose* semantics which does not require the rigidity of the specifications with *initial* denotations), which becomes particularly clear when doing proofs.

```
mod* LIST {
  protecting(TRIV+)
  *[ List ]*
  op nil : -> List
  op cons : Elt List -> List   {coherent} - - action
  bop car : List -> ?Elt       - - observation
  bop cdr : List -> List       - - action
  vars E E' : Elt
  var L : List
  eq car(nil) = err .
  eq car(cons(E, L)) = E .
  beq cdr(nil) = nil .
  beq cdr(cons(E, L)) = L .
}
```

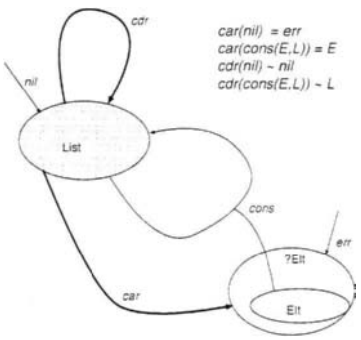
Here lists are regarded as objects (more precisely as states of a list object, hence the hidden sort List) parameterized over any set by using an extension of the built-in module TRIV with an error element:

```
mod! TRIV+ (X :: TRIV) {
  op err :  -> ?Elt
}
```

where ?Elt is a built-in error supersort of Elt.

The list object has *car* as observation (attribute) and *cdr* as action (method), while *cons* is specified as a *hidden constructor* by the coherence declaration. The fact that *cons* is not specified as a behavioural operation has the advantage of simplifying the definition of behavioural equivalence (by not involving *cons* in the behavioural contexts), the actual relation being the same (because *cons* is behaviourally coherent). We provide the proof score for the coherence of *cons* later. Also notice that in this specification we do not use any sort for the non-empty lists, the errors being handled by an error super-sort (?Elt) for the data.

This specification can be represented graphically by using the following extension of the ADJ diagrammatic notation for the CHA formalism:



The following conventions are used in the above diagram:

- *Sorts are represented by ellipsoidal disks with visible (data) sorts represented in white and hidden (state) sorts represented in grey, and with subsort inclusion represented by disk inclusion, and*
- *Operations are represented by multi-source arrows with the monadic part from the hidden sort thickened in case of behavioural operations.*

Now we concentrate for a while on expressing the behavioural equivalence relation for lists and on proving the coherence of *cons*. We need some user-defined naturals; let's consider a very simple specification of natural numbers:

```
mod! BARE-NAT {
  [ NzNat Zero < Nat ]
  op 0 : -> Zero
  op s_ : Nat -> NzNat
}
```

This is a specification with *initial* denotation whose model (unique modulo isomorphisms) consists exactly of the natural numbers with successor function (s(_)).
The following module defines a second order cdr function:

```
mod* LIST* {
  protecting(LIST + BARE-NAT)
  bop cdr : Nat List -> List
  eq cdr(0, L:List) = L .
  eq cdr(s(N:Nat), L:List) = cdr(N, cdr(L)) .
}
```

The behavioral equivalence on lists can be defined as

$$l \sim l' \text{ iff } car(cdr^n(l)) = car(cdr^n(l')) \text{ for all natural numbers } n.$$

In CafeOBJ we can handle the universal quantifiers over the natural numbers and over the set of elements by parameterization as follows:

```
mod* LIST-BEQ {
  protecting(LIST* + BARE-NAT)
  op _R[_]_ : List Nat List -> Bool {coherent}
```

```
  eq L:List R[N:Nat] L':List = car(cdr(N, L)) == car(cdr(N, L')) .
}
```

Now we can proceed with the proof score for the coherence of *cons*.

```
open .
 op n :  -> Nat .
 ops l l' :  -> List .
 ops e e' :  -> Elt .
```

The following hypothesis states that the lists l and l' are behaviourally equivalent. Its use in CafeOBJ computations is subject to a condition (which amounts to the novel concept of *behavioural rewriting* of [8-10]) which ensures the correctness of computations with behavioural equations; this is a special feature of the CafeOBJ execution mechanism.

```
 beq l = l' .
```

Now we proceed with the actual proof of the coherence of *cons*. The proof involves a small case analysis:

```
red cons(e, l) R[0]   cons(e, l') .
red cons(e, l) R[s n] cons(e, l') .
close
```

Recently, Bidoit and Hennicker [1] have formulated a syntactic sufficient criterion for the behavioural coherence property, which is relevant for most examples, including the behavioural lists. This criterion can be automatically checked by the system, thus in many cases saving such coherence proofs.

4.2. Behavioural Sets

Now we define behavioural sets and show they can be implemented by behavioural lists. Let's first start by defining basic sets:

```
mod* BASICSETS (X :: TRIV) {
 protecting(PROPC)
 *[ Set ]*
 op empty : -> Set
 op add   : Elt Set -> Set  {coherent}
 bop _in_ : Elt Set -> Bool
 vars E E' : Elt
 var S : Set
 eq E in add(E', S) = (E == E') or (E in S) .
 eq E in empty = false .
}
```

Here we use the built-in module PROPC which extends BOOL with more Boolean operations, transforming it into a decision procedure for propositional calculus.

The behavioural equivalence relation is the default coinduction relation =*= because there is only one behavioural operation, namely *_in_*. We skip here the proof of the coherence *add*, which can be done in a similar manner to the proof of coherence of *cons* for the behavioural lists.

We concentrate now on proving that LIST is a refinement of BASICSETS via the signature morphism mapping

- the hidden sort Set to List,

- the constant *empty* to *nil*,

- the behaviourally coherent operation *add* to *cons*, and

- the observation *E in L* to the derived (behavioural) operation
  (E == car(L)) or-else (car(L) =/= err and-also E in cdr(L)).

We use here *and-also* and *or-else* rather than the usual BOOL-connectives *and* and *or* in order to avoid non-termination of computations. The difference between the former and the latter is purely operational, in the sense that, for example, *and-also* evaluates the first argument first, and if this is false, then it does not evaluate the second argument anymore.

The following is a simple proof score that this mapping is a refinement specification morphism:

```
mod* LIST' {
  protecting(LIST)
  op _in_ : Elt List -> Bool  {coherent}
  vars E E' : Elt
  var L : List
  eq E in L = (E == car(L)) or-else (car(L) =/= err  and-also  E in cdr(L)) .
}
open LIST' .
  ops e e1 e2 : -> Elt .
  op l : -> List .
```

The following are just the basic cases to be analyzed:

```
  eq e1 in l = true .
  eq e2 in l = false .
```

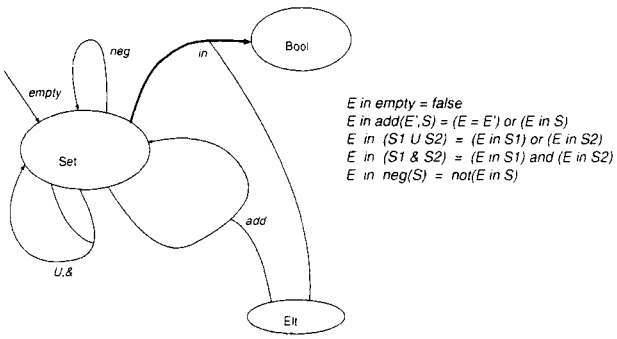
and the following are the corresponding reductions:

```
red e  in nil      == false .
red e1 in cons(e,l) == true  .
red e2 in cons(e,l) == false .
red e  in cons(e,l) == true  .
close
```

The next step is to enrich BASICSETS with new set-theoretical operations, such as union (*_U_*), intersection (*_&_*), and negation (*not*).

```
mod* SETS {
  protecting(BASICSETS)
  op _U_ : Set Set -> Set  {coherent}
  op _&_ : Set Set -> Set  {coherent}
  op not : Set -> Set      {coherent}
  var E : Elt
  vars S1 S2 : Set
  eq E in S1 U S2  = (E in S1)  or (E in S2) .
  eq E in S1 & S2  = (E in S1) and (E in S2) .
  eq E in not(S1)  = not (E in S1) .
}
```

The graphical representation of the SETS specification is as follows:



The union, intersection, and negation are behaviourally coherent, however we skip here the corresponding proofs.

Notice that the first two operations can be easily implemented in LIST, for example the union can be implemented as *append* on lists.

The usual set-theoretic properties of union, intersection, and negation can be now proved as behavioural properties:

```
open .
  op e : -> Elt .
  ops s1 s2 s3 : -> Set .
```

The following can be used again by the Theorem of Constants:

```
  ceq S1 =*= S2 = true if (e in S1) == (e in S2) .
```

Commutativity of union:

```
  red (s1 U s2) =*= (s2 U s1) .
```

Associativity of union:

```
  red (s1 U (s2 U s3)) =*= ((s1 U s2) U s3) .
```

Idempotency of union:

```
  red (s1 U s1) =*= s1 .
```

empty is the unity for the union:

```
  red (empty U s1) =*= s1 .
```

Commutativity of intersection:

```
  red (s1 & s2) =*= (s2 & s1) .
```

Associativity of intersection:

```
  red (s1 & (s2 & s3)) =*= ((s1 & s2) & s3) .
```

Idempotency of intersection:

```
  red (s1 & s1) =*= s1 .
```

empty intersected with anything is *empty*:

```
  red (empty & s1) =*= empty .
```

Distributivity:

```
  red (s1 & (s2 U s3)) =*= ((s1 & s2) U (s1 & s3)) .
  red (s1 U (s2 & s3)) =*= ((s1 U s2) & (s1 U s3)) .
```

De Morgan laws:

```
  red not(s1 U s2) =*= (not(s1) & not(s2)) .
  red not(s1 & s2) =*= (not(s1) U not(s2)) .
```

and finally, the double negation law:

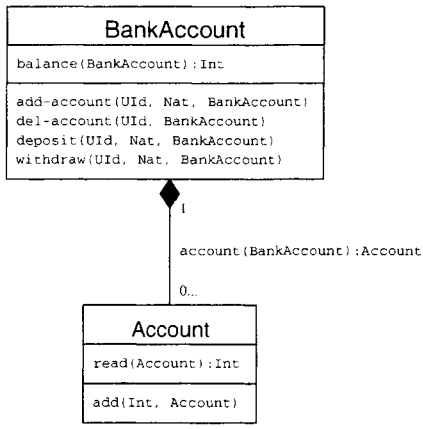
red *not(not(s1)) =*= s1* .
close

At the end of this non-conventional journey through the realm of lists and sets, the reader may try to compare the behavioural approach on sets with the data-oriented one. The advantage of the behavioural approach can be seen even from the specifications, however this becomes evident when doing proofs. The simplicity of behavioural proofs contrasts sharply with the complex and tedious proofs by induction which are implied by the data-oriented approach. This meets somehow the software engineering tradition and mathematical intuition in which sets have a clear behavioural flavour.

5. Composing Objects Concurrently

The object-oriented method in CafeOBJ [8,10,23] supports concurrent object composition within the behavioural specification approach by using the so-called *projection operations* [8,10,23] from the hidden sort of the compound object to the hidden sorts of the components. We distinguish two kinds of objects: *static* and *dynamic*. Dynamic objects can be created or deleted during the runs of the system, while the static objects cannot. By using some object-oriented terminology, we may say that dynamic objects form a class, while the static objects represent the particular case in which the class contains only one object.

As example, consider a bank accounts system consisting of several individual accounts. In UML notation this may be represented by the the following figure.



The specification of personal accounts can be considered as a dynamic system of counters of integers. The following specifies the class of the counters, with the counter identifiers parameterized by *X*.

mod* COUNTER(*X* :: TRIV) {
protecting(INT)
[Counter]
op *init-counter* : Elt -> Counter
op *no-counter* : -> Counter

```
  bop add : Int Counter -> Counter
  bop amount_ : Counter -> Nat
  var ID : Elt
  var I : Int
  var C : Counter
  beq add(I, no-counter) = no-counter .
  eq amount init-counter(ID) = 0 .
  ceq amount add(I, C) = I + amount(C) if I + amount(C) >= 0 .
  ceq amount add(I, C) = amount(C) if I + amount(C) < 0 .
}
```

The space of the states of a counter is represented by the hidden sort Counter. The action *add* counts up or down (if a negative number is passed as argument) and we can observe the state of a counter by using the observation *amount_*. This counter allows the application of the action *add* only when the observation of the resulting state is positive. This means that no substraction which would result in a negative result is allowed.

The counter is represented as a dynamic object and we require an identifier for specifying an individual counter. The operation *no-counter* is used for error situations, i.e. when some other objects specify a wrong identifier (meaning the object does not exist).

Accounts are just a renaming of the COUNTER instantiated by USER-ID (for identifiers).

```
mod* USER-ID {
  [ Uid ]
}
mod* ACCOUNT {
  protecting(COUNTER(X <= view to USER-ID { sort Elt -> Uid })
                *{ hsort Counter -> Account,
                   op init-counter -> init-account,
                   op no-counter -> no-account })
}
```

Now we can compose the account objects to form a dynamic system of bank accounts:

```
mod* ACCOUNT-SYS {
  protecting(ACCOUNT)
  *[ AccountSys ]*
  op init-account-sys : -> AccountSys
  bop add-account : Uid Nat AccountSys -> AccountSys
  bop del-account : Uid AccountSys -> AccountSys
  bop deposit : Uid Nat AccountSys -> AccountSys
  bop withdraw : Uid Nat AccountSys -> AccountSys
  bop balance : Uid AccountSys -> Nat
  bop account : Uid AccountSys -> Account
  vars U U' : Uid
  var A : AccountSys
  var N : Nat
  eq account(U, init-account-sys) = no-account .
  ceq account(U, add-account(U', N, A)) = add(N, init-account(U))      if U == U' .
  ceq account(U, add-account(U', N, A)) = account(U, A)      if U =/= U' .
  ceq account(U, del-account(U', A)) = no-account          if U == U' .
  ceq account(U, del-account(U', A)) = account(U, A)      if U =/= U' .
```

```
  ceq account(U, deposit(U', N, A)) = add(N, account(U, A))        if U == U' .
  ceq account(U, deposit(U', N, A)) = account(U, A)                 if U =/= U' .
  ceq account(U, withdraw(U', N, A)) = add(-(N), account(U, A))     if U == U' .
  ceq account(U, withdraw(U', N, A)) = account(U, A)                if U =/= U' .
  eq balance(U, A) = amount account(U, A) .
}
```

Notice the following steps involved in specifying a compound object:

1. import the component objects (ACCOUNT),
2. define a new hidden sort for the compund object and a (behavioural) projection operation to the hidden sort of each of the components (*account*),
3. define actions on the compound object (*withdraw*, *deposit*) corresponding to the component objects actions and express the relationship between compound object actions and components objects actions by (strict) equations, and
4. (eventually) define some observations on the compound objects as abbreviations for component objects observations (*balance*).

In the initial state (*init-account-sys*), the system of bank accounts does not contain any individual account. The operation *add-acount* add individual accounts and *del-account* deletes individual accounts.

The projection operations are subject to several precise technical conditions which are mathematically formulated in [8,10,23].

5.1. Proving Behavioural Properties for the Composed Object

One of the advantages of this methodology for composing objects is not only the high reusability of the specification code, but also of verification proofs. This is supported by the follwing:

Theorem 5.1 *[23] The behavioural equivalence of the compound object is the conjunction via the projection operations of the behavioural equivalences of the component objects.*

This means that the behavioural equivalence for the compound object is obtained directly without having to do any proofs about it (it is enough to establish the behavioural equivalence at the level of the composing objects). In the case of a hierarchical object composition process (involving several levels of composition), this may prove very effective in practice.

Coming back to our example, let us first look at the behavioural equivalence of ACCOUNT. This is exactly the default coinduction relation $=^*=$, but due to the lack of automatic bulit-in case analysis the CafeOBJ system fails to prove it. So, let us do it by ourselves:

```
mod BEQ-ACCOUNT {
  protecting(ACCOUNT)
  op _=*=_ : Account Account -> Bool {coherent}
  vars A1 A2 : Account
```

```
  eq A1 =*= A2 = amount(A1) == amount(A2) .
}
open BEQ-ACCOUNT .
  ops a1 a2 : -> Account .
  op i : -> Int .
```

The hypothesis for the first case is:

```
  eq i + amount(a2) >= 0 = true .
  eq amount(a1) = amount(a2) .
```

and this is the proof:

```
  red add(i, a1) =*= add(i, a2) .
  close
```

The second case is very similar:

```
  open BEQ-ACCOUNT
    ops a1 a2 : -> Account .
    op i : -> Int .
    eq i + amount(a2) < 0 = true .
    eq amount(a1) = amount(a2) .
    red add(i, a1) =*= add(i, a2) .
  close
```

Now, we obtain the behavioural equivalence of ACCOUNT-SYS by reusing the behavioural equivalence $=*=$ of ACCOUNT:

```
  mod BEQ-ACCOUNT-SYS {
    protecting(BEQ-ACCOUNT)
    protecting(ACCOUNT-SYS)
    op _R[_]_ : AccountSys Uid AccountSys -> Bool  {coherent}
    vars BA1 BA2 : AccountSys
    var U : Uid
    eq BA1 R[ U ] BA2 = account(U, BA1) =*= account(U, BA2) .
  }
```

Notice that the behavioural equivalence of ACCOUNT-SYS is parameterized by Uid; this is because it is an infinite conjunction of the behavioural equivalences for the individual accounts:

$$ba1 \sim_{\text{ACCOUNT-SYS}} ba2 \quad \text{iff} \quad account(u, ba1) \sim_{\text{ACCOUNT}} account(u, ba2)$$

for all user identifies u, where $\sim_{\text{ACCOUNT-SYS}}$ and $\sim_{\text{ACCOUNT}}$ are the behavioural equivalences of ACCOUNT and ACCOUNT-SYS, respectively.

Consider the following very simple behavioural property:

$$withdraw(u, n, ba) \sim ba$$

if there is no account corresponding to the user identifier u, where n is a natural value and ba is any state of the acounts system, meaning basically the protection of the individual accounts of the bank account system from outsiders.

```
  open BEQ-ACCOUNT-SYS
    op ba : -> AccountSys .
    ops u u' : -> Uid .
    op n : -> Nat .
```

Here is the working hypothesis:

```
eq account(u, ba) = no-account .
```

and this is the proof by very simple case analysis (whether the parameter is the same or not with the user identifier):

```
red withdraw(u, n, ba) R[ u ] ba .
red withdraw(u, n, ba) R[ u' ] ba .
close
```

A more sophisticated behavioural property is the true concurrency of the actions of two different users, which assures that the individual accounts can be safely distributed. The true concurrency properties are expressed as behavioural commutativity properties [16], such as the following one:

$$withdraw(u_1, n_1, withdraw(u_2, n_2, ba)) \sim withdraw(u_2, n_2, withdraw(u_1, n_1, ba))$$

where u_1 and u_2 are users (identifiers), and n_1 and n_2 are amounts of cash requested for withdrawal.

Now we proceed with the proof of this property:

```
mod PROOF {
  protecting(BEQ-ACCOUNT-SYS)
  op ba : -> AccountSys
  ops u u1 u2 : -> Uid
  ops n1 n2 n01 n02 m1 m1' m2 m2' : -> Nat
```

The last constants are used for the automatic generation of cases by CafeOBJ, the *n*s standing for the amount of cash requested and the *m*s for the balance of the corresponding individual accounts at the moment of the withdrawal request. The following describe all possible atomic cases of relationships between the *n*s and the *m*s:

```
  eq n1 =/= 0 = true .
  eq n2 =/= 0 = true .
  eq n01 == 0 = true .
  eq n02 == 0 = true .
  eq n1 <= m1 = true .
  eq n01 <= m1 = true .
  eq n1 > m1' = true .
  eq n2 <= m2 = true .
  eq n02 <= m2 = true .
  eq n2 > m2' = true .
```

The following declarations build the proof term (*RESULT*):

```
  op state-of-system : Nat Nat -> AccountSys
  ops W1W2 W2W1 : Nat Nat Nat Nat -> AccountSys
  op TERM : Uid Nat Nat Nat Nat -> Bool
  op TERM1 : Uid Nat Nat -> Bool
  op TERM2 : Uid -> Bool
  op RESULT : -> Bool
  var U : Uid
  vars N1 N2 M1 M2 : Nat
```

The following just says that the balance available for u_1 is M_1 and the balance available for u_2 is M_2:

```
trans state-of-system(M1, M2) => add-account(u1, M1, add-account(u2,M2, ba)) .
```

Now we build the initial proof term:

```
trans W1W2(N1,N2,M1,M2) => withdraw(u1,N1,withdraw(u2,N2,state-of-system(M1,M2))) .
trans W2W1(N1,N2,M1,M2) => withdraw(u2,N2,withdraw(u1,N1,state-of-system(M1,M2))) .
trans TERM(U,N1,N2,M1,M2) => W1W2(N1,N2,M1,M2) R[U] W2W1(N1,N2,M1,M2) .
```

And now we generate the final proof term by orthogonal instantiation of atomic cases to concrete cases:

```
trans TERM1(U, N2, M2) => TERM(U, n1, N2, m1, M2) and
                          TERM(U, n1, N2, m1', M2) and
                          TERM(U, n01, N2, m1, M2) .
trans TERM2(U) => TERM1(U, n2, m2) and
                  TERM1(U, n2, m2') and
                  TERM1(U, n02, m2) .
trans RESULT => TERM2(u) and TERM2(u1) and TERM2(u2) .
```

This amounts to 27 cases which are generated by the CafeOBJ system. Notice that this method for dealing with complex case analysis (which is inevitable when verifying large systems), besides being automatic, it can also be used very effectively for debugging. For this, one has just to use partial proof terms such as *TERM2*, *TERM1*, and *TERM* with various values as arguments in a top-down stepwise manner.

Finally we ask the CafeOBJ system to do the following reduction:

```
exec RESULT .
```

and get true as result. This means our (formal) proof of the true concurrency of withdrawals by two different users is completed.

Notice also the use of the RWL paradigm for the meta-level encoding of the case analysis which can be regarded as RWL encoding of the algorithm generating the proof.

5.2. Synchronization

In the bank accounts example, there is no synchronization between the components. In this final section we show how synchronization can be handled by using conditional equations (whose conditions must obey some rules; for precise details see [8,10,23]).

Let us consider again the example of counter, this time as a static object.

```
mod* COUNTER {
  protecting(INT)
  *[ Counter ]*
  op init-counter : -> Counter
  bop add : Int Counter -> Counter
  bop amount_ : Counter -> Int
  var I : Int
  var C : Counter
  eq amount(init-counter) = 0 .
  eq amount(add(I, C)) = I + amount(C) .
}
```

Now we put a switch to the counter, such that if the switch is on then the counter counts up, and when the switch is off then the counter counts down.

```
mod! ON-OFF {
  [ Value ]
```

```
  ops on off : -> Value
 }
 mod* SWITCH {
  protecting(ON-OFF)
  *[ Switch ]*
  op init-switch : -> Switch
  bop on : Switch -> Switch
  bop off : Switch -> Switch
  bop state : Switch -> Value
  var S : Switch
  eq state(init-switch) = off .
  eq state(on(S)) = on .
  eq state(off(S)) = off .
 }
```

The following is the specification of the counter with switch. The synchronization appears in the (conditional) equations [c-2] and [c-3].

```
 mod* COUNTER-WITH-SWITCH {
  protecting(SWITCH + COUNTER)
  *[ Cws ]*
  op init : -> Cws
  bop put : Int Cws -> Cws
  bop add : Cws -> Cws
  bop sub : Cws -> Cws
  bop amount : Cws -> Int
  bop counter : Cws -> Counter
  bop switch : Cws -> Switch
  var N : Int
  var C : Cws
  eq amount(C) = amount(counter(C)) .
  eq [s-1] : switch(init) = init-switch .
  eq [s-2] : switch(put(N, C)) = switch(C) .
  eq [s-3] : switch(add(C)) = on(switch(C)) .
  eq [s-4] : switch(sub(C)) = off(switch(C)) .
  eq [c-1] : counter(init) = init-counter .
  cq [c-2] : counter(put(N, C)) = add(N, counter(C))
             if state(switch(C)) == on .
  cq [c-3] : counter(put(N, C)) = add(-(N), counter(C))
             if state(switch(C)) == off .
  eq [c-4] : counter(add(C)) = counter(C) .
  eq [c-5] : counter(sub(C)) = counter(C) .
 }
```

Notice the (hidden) sort of the compound object is Cws, and the projection operations are *switch* and *counter*. The behavioural equivalence of COUNTER-WITH-SWITCH is conjunction of the behavioural equivalences of SWITCH and COUNTER:

```
 mod BEQ-CWS {
  protecting(COUNTER-WITH-SWITCH)
  op _R_ : Cws Cws -> Bool
```

```
  vars C1 C2 : Cws
  eq  C1 R C2 = counter(C1) =*= counter(C2)   and   switch(C1) =*= switch(C2) .
}
```

We can prove, for example, the following behavioural property:

$$put(n, add(put(n', sub(c)))) \sim add(put(n', sub(put(n, add(c)))))$$

```
open BEQ-CWS
 op c : -> Cws .
 ops n n' : -> Nat .
red put(n, add(put(n', sub(c)))) R add(put(n', sub(put(n, add(c))))) .
close
```

REFERENCES

1. Michel Bidoit and Rolf Hennicker. Observer complete definitions are behaviourally coherent. In K. Futatsugi, J. Goguen, and J. Meseguer, editors, *OBJ/CafeOBJ/Maude at Formal Methods '99*, pages 83–94. Theta, Bucharest, 1999.
2. Rod Burstall and Joseph Goguen. The semantics of Clear, a specification language. In Dines Bjorner, editor, *Proceedings, 1979 Copenhagen Winter School on Abstract Software Specification*, pages 292–332. Springer, 1980. Lecture Notes in Computer Science, Volume 86.
3. Manuel Clavel, Steve Eker, Patrick Lincoln, and Jose Meseguer. Principles of Maude. *Electronic Notes in Theoretical Computer Science*, 4, 1996. Proceedings, First International Workshop on Rewriting Logic and its Applications. Asilomar, California, September 1996.
4. Răzvan Diaconescu. Category-based semantics for equational and constraint logic programming, 1994. DPhil thesis, University of Oxford.
5. Răzvan Diaconescu. Extra theory morphisms for institutions: logical semantics for multi-paradigm languages. *J. of Applied Categorical Structures*, 6(4):427–453, 1998. A preliminary version appeared as JAIST Technical Report IS-RR-97-0032F in 1997.
6. Răzvan Diaconescu and Kokichi Futatsugi. Logical foundations of CafeOBJ. 1998. Submitted to publication.
7. Răzvan Diaconescu and Kokichi Futatsugi. Logical semantics for CafeOBJ. In *Precise Semantics for Software Modeling Techniques*, pages 31–54. Proceedings of an ICSE'98 workshop held in Kyoto, Japan, published as Technical Report TUM-I9803, Technical University Munchen, 1998. Preliminary version appeared as Technical Report IS-RR-96-0024S at Japan Advanced Institute for Science and Technology in 1996.
8. Răzvan Diaconescu and Kokichi Futatsugi. *CafeOBJ Report: The Language, Proof Techniques, and Methodologies for Object-Oriented Algebraic Specification*, volume 6 of *AMAST Series in Computing*. World Scientific, 1998.
9. Răzvan Diaconescu and Kokichi Futatsugi. Behavioural coherence in object-oriented algebraic specification. *J. of Universal Computer Science*, 2000. First version appeared as JAIST Technical Report IS-RR-98-0017F, June 1998.
10. Răzvan Diaconescu, Kokichi Futatsugi, and Shusaku Iida. Component-based algebraic specification and verification in CafeOBJ. In Jeannette M. Wing, Jim Woodcock,

and Jim Davies, editors, *FM'99 - Formal Methods*, volume 1709 of *Lecture Notes in Computer Science*, pages 1644–1663. Springer, 1999.

11. Răzvan Diaconescu, Joseph Goguen, and Petros Stefaneas. Logical support for modularisation. In Gerard Huet and Gordon Plotkin, editors. *Logical Environments*, pages 83–130. Cambridge, 1993. Proceedings of a Workshop held in Edinburgh, Scotland, May 1991.
12. Kokichi Futatsugi, Joseph Goguen, Jean-Pierre Jouannaud, and Jose Meseguer. Principles of OBJ2. In *Proceedings of the 12th ACM Symposium on Principles of Programming Languages*, pages 52–66. ACM, 1985.
13. Joseph Goguen. *Theorem Proving and Algebra*. MIT, 2000. To appear.
14. Joseph Goguen and Rod Burstall. Institutions: Abstract model theory for specification and programming. *Journal of the Association for Computing Machinery*, 39(1):95–146, January 1992.
15. Joseph Goguen and Răzvan Diaconescu. An Oxford survey of order sorted algebra. *Mathematical Structures in Computer Science*, 4(4):363–392, 1994.
16. Joseph Goguen and Răzvan Diaconescu. Towards an algebraic semantics for the object paradigm. In Harmut Ehrig and Fernando Orejas, editors, *Recent Trends in Data Type Specification*, volume 785 of *Lecture Notes in Computer Science*, pages 1–34. Springer, 1994.
17. Joseph Goguen and Grant Malcolm. A hidden agenda. Technical Report CS97-538, University of California at San Diego, 1997.
18. Joseph Goguen and José Meseguer. Eqlog: Equality, types, and generic modules for logic programming. In Douglas DeGroot and Gary Lindstrom, editors, *Logic Programming: Functions, Relations and Equations*, pages 295–363. Prentice-Hall, 1986. An earlier version appears in *Journal of Logic Programming*, Volume 1, Number 2, pages 179–210, September 1984.
19. Joseph Goguen and José Meseguer. Unifying functional, object-oriented and relational programming, with logical semantics. In Bruce Shriver and Peter Wegner, editors, *Research Directions in Object-Oriented Programming*, pages 417–477. MIT, 1987. Preliminary version in *SIGPLAN Notices*, Volume 21, Number 10, pages 153–162, October 1986.
20. Joseph Goguen and José Meseguer. Order-sorted algebra I: Equational deduction for multiple inheritance, overloading, exceptions and partial operations. *Theoretical Computer Science*, 105(2):217–273, 1992.
21. Joseph Goguen, Timothy Winkler, José Meseguer, Kokichi Futatsugi, and Jean-Pierre Jouannaud. Introducing OBJ. In Joseph Goguen and Grant Malcolm, editors, *Software Engineering with OBJ: algebraic specification in action*. Kluwer, 2000. To appear.
22. Rolf Hennicker and Michel Bidoit. Observational logic. In A. M. Haeberer, editor, *Algebraic Methodology and Software Technology*, number 1584 in LNCS, pages 263–277. Springer, 1999. Proc. AMAST'99.
23. Shusaku Iida, Kokichi Futatsugi, and Răzvan Diaconescu. Component-based algebraic specification: - behavioural specification for component-based software engineering -. In *Behavioral specifications of businesses and systems*, pages 103–119. Kluwer, 1999.
24. Donald Knuth. *The Art of Computer Programming: Sorting and Searching*. Addison-

Wesley, 1973.

25. José Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.
26. José Meseguer. Membership algebra as a logical framework for equational specification. In F. Parisi-Pressice, editor, *Proc. WADT'97*, number 1376 in Lecture Notes in Computer Science, pages 18–61. Springer, 1998.

Chapter 3

An Overview of the Tatami Project*

Joseph Goguen[a] and Kai Lin[a] and Grigore Roşu[a] and Akira Mori[b], and Bogdan Warinschi[a]

[a] Dept. Computer Science & Engineering, University of California, San Diego
9500 Gilman Drive, La Jolla CA 92093-0114 USA
phone: +1-858-822-1588, fax: +1-858-534-7029
email: {goguen,klin,grosu,bogdan}@cs.ucsd.edu

[b] Japan Advanced Instititute of Science and Technology, Hokuriku
1-1 Asahidai Tatsunokuchi Nomi Ishikawa, 923-1292 JAPAN
email: amori@jaist.ac.jp

This paper describes the Tatami project at UCSD, which is developing a system to support distributed cooperative software development over the web, and in particular, the validation of concurrent distributed software. The main components of our current prototype are a proof assistant, a generator for documentation websites, a database, an equational proof engine, and a communication protocol to support distributed cooperative work. We believe behavioral specification and verification are important for software development, and for this purpose we use first order hidden logic with equational atoms. The paper also briefly describes some novel user interface design methods that have been developed and applied in the project.

1. Introduction

The Tatami system design was motivated by three main goals: (1) verify distributed concurrent software; (2) provide a distributed environment for cooperative work; and (3) produce proofs that are easier to read. This system differs from related systems with which we are familiar, in many or all of the following respects:

1. it is rigorously based on an advanced version of hidden algebra allowing first order sentences with equational atoms interpreted behaviorally;
2. design is separated from validation, with a distinct language for each activity;
3. distributed cooperative work is supported;
4. there is a distributed multi-project database;
5. a specialized protocol maintains database consistency in the presence of semi-reliable internet communications;

*The research reported in this paper has been supported in part by the CafeOBJ project of the Information Promotion Agency (IPA), Japan, as part of its Advanced Software Technology Program, and by National Science Foundation grant CCR-9901002.

6. a range of formality is supported from full mechanical proofs to informal "back of envelope" arguments, using fuzzy logic for the confidence values of assertions;
7. web-based interactive documentation is automatically generated for proofs;
8. there are links to online tutorials on background material;
9. recent web and net technologies are heavily used, including secure HTTP, XML [40], XSL [41], SSL [39], and cgi; and
10. many user interface design decisions have been rigorously based on principles from cognitive psychology, narratology, semiotics, etc.

Large software projects have multiple workers, often at multiple sites with different schedules. Therefore it is difficult to share information, coordinate tasks, and maintain consistent code, documentation and requirements. We seek to solve some of these problems by building tools to support distributed cooperative work over the web. Our current emphasis is on proving properties of distributed concurrent systems, such as communication protocols. Although full formal verification is an option, the most practical approach is probably to exploit the task structure of formal methods without the burden of providing complete formal proofs for all steps; in this way, formality provides a discipline both for designing tools and for using them.

We focus on behavioral specification and verification, because real software systems often do not satisfy their specifications strictly, but instead only satisfy them *behaviorally*, i.e., appear to satisfy them in all relevant situations. Our logic for this is hidden algebra [19,35]. Code is regarded as secondary, because it can be (relatively!) quickly written or even automatically generated from specifications that are sufficiently modular and detailed, and empirical studies show that little of software cost comes from errors in coding [3]. Therefore we focus on specification and verification at the design level, and avoid the ugly complications of programming language semantics.

Since we wish to assist ordinary software engineers in using formal methods to design and verify complex systems, an important task for our project is to find better ways to explain and document proofs. Examining any mathematics book or paper, even one in logic, shows that mathematicians almost never write formal proofs in the strict sense of mathematical logic. The only proofs written this way are extremely simple illustrations of formal proof methods, not proofs of any genuine mathematical interest. This is because all but the simplest fully formal proofs are practically impossible to comprehend. Unfortunately, these are exactly the kind of proofs produced by mechanical theorem provers. In order to improve this dismal situation, we suggest that the motivation and structure of proofs should be made explicit, and that relevant background and tutorial material should be integrated with proofs. These recommendations follow ideas from cognitive psychology, narratology and semiotics [11,12], as discussed further in Section 3. In particular, the structure of the Tatami system proof websites was designed using algebraic semiotics, which combines algebraic specification with social semiotics in the sense of [10]. The present paper is an extension, revision and amalgamation of work reported in [15,16] and other papers. Although this particular paper focuses on system architecture and user interface issues, for the convenience of interested readers, the bibliography attempts to list most of the papers produced by the project, including those concerning other aspects, such as the underlying formal logic. The latest information on the Tatami project can always be accessed via its URL, `www.cs.ucsd.edu/groups/tatami/`.

2. Tatami System Design

This section describes the Tatami System, including its central component, a proof assistant and website generator component KUMO[1], its specification language and underlying hidden logic (Section 2.1), its command language (Section 2.1), database (Section 2.2), and its specialized communication protocol (Section 2.3). Some further implementation details are given in Section 2.4.

2.1. KUMO

KUMO executes proof commands, and generates websites that document its proofs. Users of Tatami mainly interact with KUMO through its database management interface and its two specialized languages, BOBJ for (behavioral) specifications, and DUCK, which has commands for both proof execution and proof display. KUMO has two user modes, called "browser" and "owner." Browser mode provides access to all of the public information, while owner mode is used for creating and editing specifications and proofs. A typical session goes as follows (see Figure 3.1):

1. Choose a project.
2. Optionally, introduce a new specification.
3. Select an existing unsolved proof task, or introduce a new proof task.
4. Enter or edit commands in a DUCK text, and then execute it; this produces or updates the corresponding proof website.
5. View the results on a web browser.
6. Repeat from step 2.
7. When done, save your work; all subtasks of the selected task are then entered into the Tatami database.

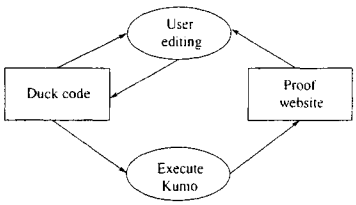


Figure 3.1. The Edit-Execute-Browse Cycle

While this edit-execute-browse cycle (again see Figure 3.1) might seem old-fashioned to some readers, empirical studies [29] and our own experience have found that the current fad for direct manipulation interfaces for theorem proving systems is actually counterproductive for complex proofs, although it may have some value when used on small

[1] This is a Japanese word for spider.

proofs as a pedagogical aid, e.g., for beginning students of formal logic. The problem is that the data structures that underlie proofs of any real interest are too large. Note also that the specification may also undergo evolution as a part of the cycle described above.

Hidden Algebra and BOBJ

The Tatami project uses an approach to behavioral specification and verification called **hidden algebra**, which extends standard many sorted algebra by distinguishing between "visible" sorts used to model data, and "hidden" sorts used to model states, as originally proposed in [9] and elaborated in many subsequent publications. This framework provides simple and natural ways to define behavioral equivalence of states, behavioral satisfaction of properties, and behavioral refinement of specifications. Standard equational deduction generalizes with small changes, but more powerful inference rules are needed for most interesting proofs. In particular, we have been developing a series of increasingly powerful **coinduction** rules [33–35], which greatly extend deductive power, and which in practice yield conceptually simple and highly mechanizable proofs; the most recent of these is called **circular coinductive rewriting** – see [34]. Recent research has shown that it is impossible to give a complete recursively enumerable set of inference rules for hidden algebra [4], so there cannot be any final resting point on this quest. For details on hidden algebra, see [18–21,34,35].

Hidden algebra handles all the main features of modern software systems, including states, classes, subclasses, attributes, methods, abstract data types, concurrency, distribution, nondeterminism, generic modules, and even logical variables (as in logic programming). In comparison with process algebra and transition systems, hidden algebra allows better control over the data involved, and admits equations with operations having multiple visible and hidden parameters, thus greatly extending expressive power.

Behavioral logic is a diverse research area containing many approaches, including the original hidden algebra of [9] and subsequent improvements in [14,19,18], the coherent hidden algebra of Diaconescu [6,7], the observational logic of Bidoit and Hennicker [1,2,24], and a new generalization of hidden algebra that tries to treat all these variants in a uniform way [35,21]. These approaches fall into two broad categories, depending on whether or not a fixed data algebra is assumed for all models. All proof rules in use are sound for all these logics, but all of them are also incomplete [4]. Padawitz's "swinging types" are a powerful but less closely related approach [31].

The BOBJ (for Behavioral OBJ) hidden algebraic specification language extends the classical algebraic specification language OBJ [23] to behavioral properties, and can be considered a dialect of the CafeOBJ specification language [7]. Like CafeOBJ, it supports both classical and hidden algebraic specification; in addition, it supports behavioral operations with multiple hidden arguments, cobases, and circular coinductive rewriting. A very brief introduction to BOBJ appears in [34]; the system has recently been implemented in Java. The Tatami system and KUMO extend the logic of BOBJ by allowing first order sentences with behavioral equations as atoms, and induction for the initially defined data types.

DUCK

DUCK contains the proof command language for KUMO, combining proof rules for hidden algebra and first order logic, including the following:

- elimination rules for $\forall$, $\exists$, $\wedge$, $\vee$, $\Rightarrow$, and $\neg$;
- Skolemization and lemma introduction;
- case analysis, modus ponens, proof by contradiction and substitution;
- term rewriting and equational deduction;
- induction; and
- coinduction.

DUCK also has a sublanguage for proof display, which provides the URLs and sets the parameters used when KUMO produces the XML file containing the tree structure of the proof. For example, when conjunction elimination is applied to the goal $T \vdash_\Sigma A \wedge B$, the two new subgoals, $T \vdash_\Sigma A$ and $T \vdash_\Sigma B$, are added to the XML file. KUMO then uses an XSL style file to generate HTML text for the proof; at this stage, links to tutorials, machine proof scores, and explanation pages are added, as described in more detail in Appendix B. DUCK also has the capability to place several proofs on a single page, and it automatically provides appropriate cross-references to other proofs used, such as lemmas.

2.2. The Tatami Database

Any practical implementation of formal methods must support bookkeeping for proof tasks, subtasks, dependencies, and specifications. Our system does this using a distributed database, the coherence of which is maintained by a specialized protocol. Each worker has a local database which is conceptually divided into the three components that are described in the three subsections below.

The Project Database

The Project Database maintains a list of projects, with their members and leaders. A project is started by placing it in the database; the worker who does so becomes the leader of that project. Information at the project database level can only be modified in owner mode by the leader of a project.

The Specification Database

The Specification Database has two major parts, one for specifications of data which are used for values, the other for abstract (software) machine specifications. A number of binary relations may be defined on specifications, including the following:

1. Importation of one specification by another.
2. Equivalence and enrichment of specifications.
3. Refinement for abstract machines.
4. Evolution from a previous version.
5. Equivalence and enrichment, for both kinds of specification.

There are also some important relations that hold among these relations, for example that enrichment implies refinement, and that equivalence implies enrichment.

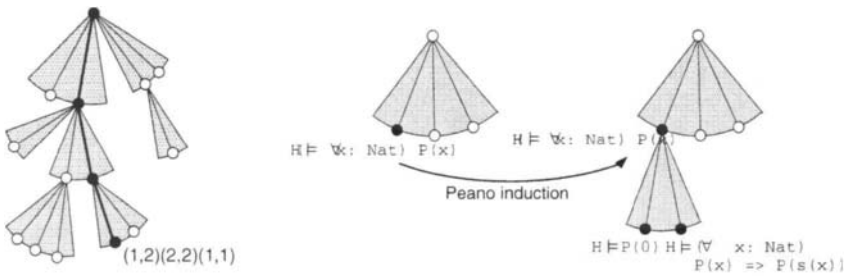


Figure 3.2. 2-occurrences and Peano induction

The Proof Database

The Proof Database keeps track of the support given for tasks and subtasks, which can range from fully formal mechanical proofs to informal "back of envelope" arguments. Alternative validations of the same task are also allowed. All this is handled by a data structure called a **2-dimensional directed ordered acyclic graph**, or **2-doag** for short. Nodes in a 2-doag are labeled with validation tasks. Instead of edges, "fans" come out of nodes, where each fan represents a validation step for the task at its source (see Figure 3.2, noting that the leftmost and rightmost lines are just to make the picture look more like a fan; they do not lead to nodes). A fan can have zero or more nodes as its targets; these represent subtasks generated by the validation step. Furthermore, there can be any number of fans coming out of any given node, representing alternative ways to validate the task on the source node.

Navigation through 2-doags is accomplished with **2-occurrences**, which are sequences of pairs of positive integers, such as (1,2)(2,2)(1,1); given a node, such a sequence points to at most one other node, by taking the first number in each pair to indicate a fan and the second a target node of that fan. Thus the above sequence says: take the second target node of the first fan, from there take the second node of the second fan, and then take the first node of the first fan, as illustrated in the left side of Figure 3.2. A formal description of 2-doags is given in Appendix C. These structures may be further specialized as follows:

- A **truthdoag** is a 2-doag with a fuzzy truth value $t(n)$ at each node n. A boolean expression defining $t(n)$ is also associated with node n; it is the disjunction of the expressions for the fans going out from n. We use the fuzzy logic of [8] to evaluate the expressions. The truth value 1 means that there is a formal proof, while 0 means that there is a formal disproof. (The logic of [8] differs from the usual fuzzy logic in using multiplication for conjunction, instead of maximum.)

- A **proofdoag** is a truthdoag such that each node has a validation task; for formal proofs, each fan corresponds to some proof rule reducing the task at the root of the fan to the tasks on its leaves. The right side of Figure 3.2 shows a proofdoag update that applies Peano induction for the natural numbers.

2.3. The Tatami Protocol

A problem with distributed databases is that local consistency can be lost if submitted proofs are inserted without being checked. For example, if a proof p_1 depends on a proof p_2, then the former should not be counted as verified in a local database unless p_2 has already been entered. Inconsistencies can similarly arise when items are deleted. The Tatami protocol maintains the consistency of the Tatami database, taking account of the following situations:

1. If the owner of an item decides to delete it, then ownership is passed to a randomly chosen worker who is using it. A message is broadcast to all involved databases to make this transfer public, and the item is deleted from its original location only after acknowledgments are received from all of them.
2. When something is missing from a just received message, the message must be retained, but not installed in the database until the missing part arrives.

Each message contains a 2-occurrence to specify the location of its update in the proofdoag. The occurrence of its parents in the sender's proofdoag are also sent, to enable detection of missing data. The message format is as follows:

1. project name;
2. a 2-occurrence for the update;
3. 2-occurrences of parents of the update; and
4. a tag indicating the kind of update, which is one of submission, deletion, copy, validation, status update, etc.

The atomic entities sent to proof databases are fans. Each message contains exactly one fan together with its position in the proofdoag. Of course, workers sometimes want to submit more than one fan; this is accomplished by dividing the desired sub-doag into fans that are sent separately. Each fan $\mathcal{F}$ is assumed to have the following attributes:

1. a unique label which gives the position of $\mathcal{F}$ in the doag (occurrences can be used as labels, noting that one fan can have multiple occurrences).
2. a message which encodes the fan $\mathcal{F}$ together with its label and all of its direct parents' labels (note that $\mathcal{F}$ can have zero, one or more parents) and the other data items discussed above.

We have used Kumo to formally prove the correctness of the Tatami protocol with respect to a communication medium that can lose or duplicate data [22], and we have implemented it using the IP internet protocol.

2.4. Some Implementation Details

Figure 3.3 is an overview of the Tatami system architecture. Its most important components are the KUMO website generator and proof assistant, the Tatami database, the Tatami protocol, the BARISTA[2] proof engine server, one or more proof engine (especially some version of OBJ), and a standard web browser. The Tatami database is implemented

[2]This is an Italian word for a person who serves espresso.

using the mSQL relational database system, and BOBJ and DUCK are implemented using Java technologies, including JavaCC for parsing. When viewing generated webpages in browser mode, a modified HTTP server is used, while owner mode uses a secure HTTP server, built using SSL.

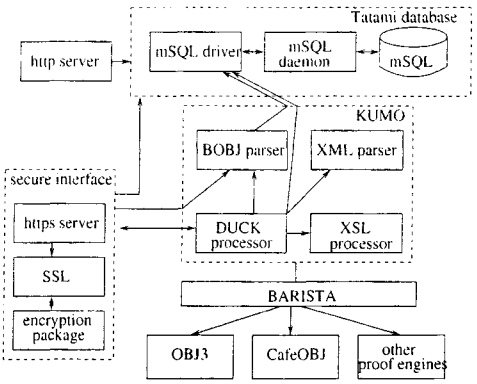


Figure 3.3. Tatami Implementation Overview

The XML files generated by KUMO are passed to a processor that uses an XSL style file to generate the HTML pages that are actually seen by users. Figure 3.4 is a UML use case diagram showing in more detail the actions that take place when a DUCK text is processed.

The Tatami project does not seek to re-implement the many complex algorithms that are provided by existing publicly available theorem proving systems. Instead, KUMO generates detailed proof scores that are sent to appropriate proof engines using the BARISTA server. Our current prototype wraps an OBJ3 proof engine, but BARISTA can easily be adapted to other proof engines, especially CafeOBJ and (very soon, we hope) BOBJ.

This modular architecture was designed to facilitate use of a variety of logics. The following components are ogic-dependent and would have to be changed to support a different logic:

- a parser for the logic's syntax;
- a "provlet" for each inference rule, which is Java code implementing that rule;
- an XSL style file containing an XML display "template" for each rule; and possibly
- a new proof engine and server.

3. User Interface Design

Because we aim to help ordinary software engineers, user interface issues are of great importance for the Tatami project. Cognitive science, semiotics (the theory of signs),

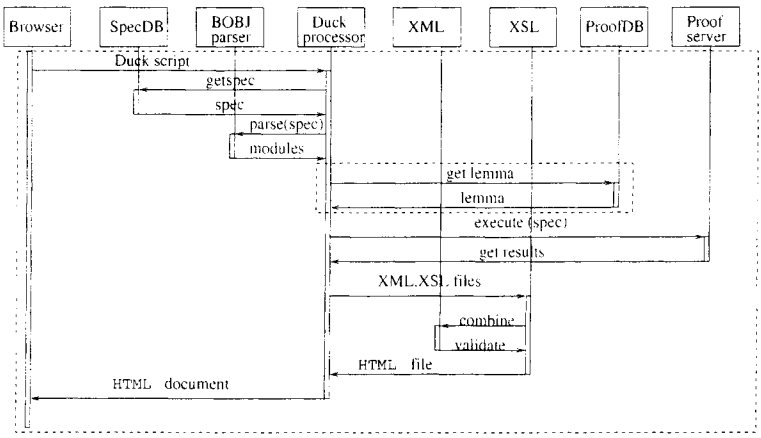


Figure 3.4. Use Case Diagram for DUCK Execution

narratology (the theory of stories) and even cinema, have influenced our designed decisions, as briefly explained below, and in much more detail in [11]. Of particular interest is algebraic semiotics, a novel technique that provides systematic ways to evaluate the quality of user interfaces, such as proof presentations.

3.1. Algebraic Semiotics

An important insight attributed to Ferdinand de Saussure [37] is that signs should not be considered in isolation, but rather as elements of *systems* of related signs, including their structural aspects. For computer scientists, it is natural to formalize the intuitive notion of a **sign system** using the tools of algebraic specification [17], as a loose algebraic theory (consisting of a signature and some axioms) plus further structure, such as constructors, a hierarchy of levels for signs, and priorities among constructors. A recent insight that is important for the Tatami project is that *representations* can be seen as translations or maps from one sign system to another [11]; it is similarly natural to formalize these translations using the notion of *theory morphism* from algebraic specification [13]. But examples in our application domains show that maps between sign systems do not in general fully preserve structure, and in particular, must involve *partial* functions. These considerations motivate the definition of **semiotic morphism** that is given in [11]. The key point for present purposes is that the *quality* of a representation can be examined in terms of what structure is preserved by its semiotic morphism.

User interfaces are of course prime examples of representations, and it is therefore natural to study them using semiotic morphisms which map from the sign system of the given information to that of its display; the quality of the interface is then measured by the quality of its semiotic morphism. Details, some of which are surprisingly technical, may be found in [12] and [11].

3.2. Narratology

Typical proofs in modern mathematics hide the often considerable conflicts that were involved in their construction, since finding a non-trivial proof usually requires exploring many misconceptions and errors, some of which may be very subtle. We claim that proofs can be made more understandable, and even more interesting, if the conflicts that motivate their difficult steps are integrated into their structure, instead of being ignored. As Aristotle said, "*Drama is conflict.*"

An important resource for our work has been the theory of oral narratives developed by William Labov [25], who showed that these have a precise structure, involving a sequence of so-called "narrative clauses" which describe events (the default ordering of which corresponds to their order in the story), interleaved with "evaluative material" which *evaluates* the events, in the sense of relating them to socially shared values. There is also an optional opening "orientation" section, giving necessary background for understanding the story, such as time and place, as well as an optional closing section containing a "moral" or summary for the story; see also [26,27]. The influence of these ideas on our proof website design conventions is discussed in in the next section.

3.3. Proofwebs and the Tatami Conventions

Proof representations can be described at several levels of abstraction. What we call a **concrete proofweb** describes our abstract structure for formal proof steps combined with informal elments; it has a tree structure with proof steps as nodes, decorated with pointers to relevant background and explanation material. An **abstract proofweb** has a similar tree structure, but omits the background and explanation material, as well as details of proof rules; only the information needed to support cooperative work is retained. Finally, a **display proofweb** is a graphical representation, consisting of windows, colors, navigation buttons, etc. Its content is the information provided by the corresponding concrete proofweb.

Professional advice for user interface design in general, and for website design in particular, nearly always calls for using style guidelines to produce a uniform "look and feel" that is appropriate for the particular application, e.g., see [32,38]. We have developed the following **tatami conventions** as style guidelines for the proof websites generated by Kumo, which we are calling display proofwebs:

1. **Tatami pages** are the most important constituents of proof websites, since they contain the inference rule applications that are the heart of proving. Tatami pages appear in a fixed master window, an example of which is shown on the right side of Figure 3.5; each page should have a relatively small number of proof steps (about seven non-automatic rules per page works well).
2. Tatami pages can be browsed in a "narrative" order, designed by the prover to be helpful and interesting to the reader; if possible, these pages should tell a story about how obstacles were overcome (or still remain).
3. A prover-supplied informal explanation page is linked to each tatami page, discussing the proof concepts, strategies, obstacles, etc. for that page; these can have graphics, applets, and of course text; they appear in a dedicated persistent popup window.

4. Major proof parts, including lemmas, each have their own homepage, which can include graphics, applets, and text; these appear in the same window as their tatami pages; see the left side of Figure 3.5. There is a dedicated persistent popup window for lemmas which has the same structure as the master window.
5. Major proof parts can have their own "closing" webpage to sum up results and lessons; these appear in the same window as their tatami pages.
6. Formal proof steps are automatically linked to pre-existing tutorial background pages; e.g., each application of induction is linked to a webpage that explains the kind of induction used. Tutorial pages also have their own dedicated persistent popup window.
7. A detailed proof score is generated for each proof page; proof readers can view these on another dedicated persistent popup, and can request their execution on a proof engine, with result displayed in the same window as the proof score.
8. Each kind of webpage has its own distinctive background and frame; the frame provides navigation buttons that are appropriate for that kind of page.
9. A menu of open subgoals is placed on each homepage, and error messages are placed on the most appropriate pages. A summary of this information also appears in the status window, which is a specialized popup that summarizes information about the current status of a proof or part thereof.

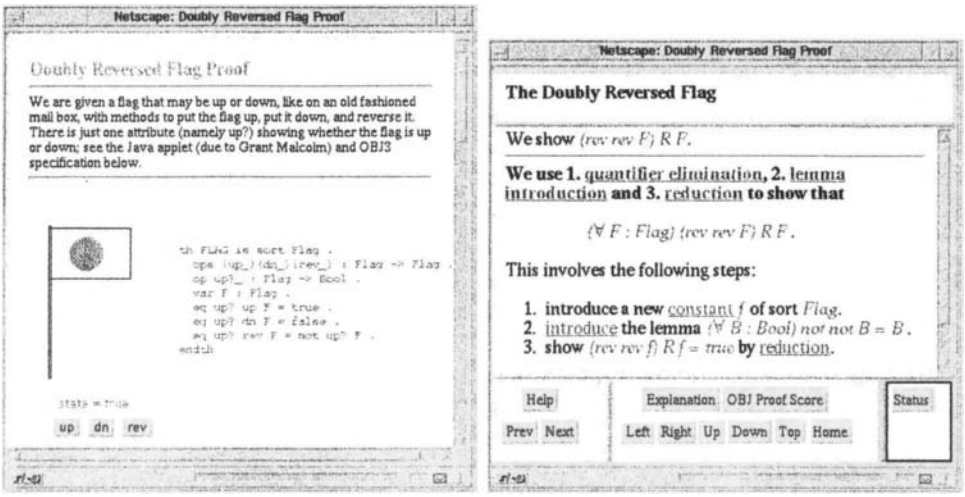


Figure 3.5. A Typical Tatami Homepage and Proof Page

These conventions have the effect of integrating proofs with the information that is needed to motivate, understand and debug them, and should therefore make proofs easier to do and easier to understand.

3.4. Justifying Design Decisions

This section applies the techniques discussed in Sections 3.1 and 3.2 to some design decisions for the proof websites generated by KUMO. We first justify the tatami conventions using the same enumeration as in Section 3.3; many of these justification draw on narratology.

1. Limiting the number of non-automatic proof steps on tatami pages to approximately 7 is consistent with known limitations of human cognitive capacity [30].
2. The idea of a giving a "narrative" order to tatami pages comes from the theory of stories [25]: the idea of including obstacles comes from Campbell [5] and others.
3. Attaching prover-supplied informal explanation pages to proof pages was suggested by the close connection between narrative clauses and evaluative material in stories [25]; the evaluative material provides the motivation needed for important steps in the proof, by relating them to values shared among the community of provers. Placing these in a separate window parallels the syntactic structure used in stories.
4. Using homepages for major proof parts is motivated by the opening "orientation" sections in Labov's theory of story structure [25]; homepages appear in the same window as their tatami pages, because they are part of the same narrative flow.
5. The optional closing webpages for proofs are also inspired by Labov's theory of story structure [25], and they too appear in the same window as proof pages, again because they are part of the same narrative.
6. Linking proof steps to tutorial pages can be motivated by some connectionist theories that concepts are organized as linked structures [36].
7. Separating mechanical proof scores from the proof pages that generate them allows hiding the most routine details of proofs, just as human proofs often omit details in order to highlight the main ideas [28]; however, proof readers can still view them, and even execute them on a proof server.
8. The justification for giving each kind of webpage a different background and a different frame is discussed below.
9. Open subgoals are very important to provers when they read a proof; they are the leaf nodes of the current proof tree, and hence form a list in a natural way.

The remaining design decisions are justified using algebraic semiotics. The basis for these arguments is that any display to users of information in the system can be seen as a semiotic morphism from the sign system for concrete proofwebs to that of display proofwebs.

- **Windows:** The main contents of an abstract proofweb are its proof steps, informal explanations, tutorials, and mechanical proof scores. These four sorts of item are also the main contents of abstract proofwebs, and their preservation is very basic to the quality of this representation. This basic classification into four sorts is reflected in our choice to use a separate window to display each of them. Because Tatami pages are the main constituent of concrete proofwebs, theirs is the master window. Explanation pages have a persistent popup window triggered by a button on the master window, and dismissable by the proof reader with a button in its frame;

tutorial and proof score pages are treated similarly, but have different colors and textures. There is also a persistent dismissable popup for lemmas, which has the exactly same structure as the master window.

- **Backgrounds:** Each kind of window has its own distinctive background: tatami pages have a tatami mat background, explanation pages have a pink marble background, tutorial pages have a yellow marble background, and proof score pages have a blue raindrop background. This again reflects the importance and distinctness of four main components of concrete proofwebs, and is justified by the importance of their preservation.

- **Frames:** A similar argument holds for frames. A top "title" frame contains the name of the current display proofweb. A "button" frame on the bottom supports navigation; these are slightly different for each kind of page, but have a uniform look and feel. The third frame holds the content, proof pages for the proof steps. Each persistent window has its own fixed layout and frames, with a button that returns to the page where it was requested.

- **Mathematical Formulae:** We use `gif` files for mathematical symbols, in a distinctive blue color, because mathematical signs come from a domain that is quite distinct from that of natural language.

Some additional, more precise, applications of semiotic morphisms to the user interface of the Tatami system are described in [12]. For example, we were able to show that certain early designs for the status window were incorrect because the corresponding semiotic morphisms failed to preserve certain key constructors. The graduate user interface design course now taught at UCSD uses ideas from algebraic semiotics (see `www.cs.ucsd.edu/users/goguen/courses/271/`), and we believe that much more can be done along these lines.

4. Conclusions and Future Research

The Tatami project has come a long way from its beginnings in 1996. Inspired by the ambitious goals of the CafeOBJ project of which it was a small part, it developed in a perhaps surprising diversity of directions, including theoretical foundations of behavioral verification for distributed concurrent systems (i.e., hidden algebra), web-based system development (the Tatami system itself), and user interface design (using algebraic semiotics). Although significant progress has been made, a very great deal still remains to be done in each of these three areas. Fortunately, they are mutually reinforcing. For example, improvements in the user interface design of the Tatami system and its KUMO prover make it easier to do proofs in hidden algebra, which in turn inspire further developments in the theory, which in turn inspire further improvements to the system.

In addition, we have new ideas for generalizing algebraic semiotics to better handle dynamic interfaces by extending its foundation from classical algebraic semantics to hidden algebra. We also plan to use the Tatami system in teaching the UCSD graduate course on programming languages, which will no doubt stimulate further developments to the system, its interfaces, and its theory. Finally, we feel we are now in position to begin

developing methodological guidelines for applying hidden algebra and the Tatami system to difficult applications like communication protocols.

Acknowledgements

We thank Prof. Kokichi Futatsugi for encouragement and support through the CafeOBJ project in Japan, Akiyoshi Sato for writing some daemons for an early prototype of the Tatami protocol, and Prof. Eric Livingston for discussions of social issues. We also thank the rapidly developing international community interested in behavioral specification and verification for encouragement.

REFERENCES

1. Gilles Bernot, Michael Bidoit, and Teodor Knapik. Observational specifications and the indistinguishability assumption. *Theoretical Computer Science*, 139(1-2):275–314, 1995. Submitted 1992.
2. Michael Bidoit and Rolf Hennicker. Observer complete definitions are behaviourally coherent. Technical Report LSV–99–4, Ecole Normale Superior de Cachan, 1999.
3. Barry Boehm. *Software Engineering Economics*. Prentice-Hall, 1981.
4. Sam Buss and Grigore Roşu. Incompletness of behavioral logics, 1999. To appear in *Proceedings, Coalgebraic Methods in Computer Science* (Berlin, Germany, 25–26 March 2000), Electronic Notes in Theoretical Computer Science, Volume 33, 2000.
5. Joseph Campbell. *The Hero with a Thousand Faces*. Princeton, 1973. Bollingen series.
6. Răzvan Diaconescu. Behavioural coherence in object-oriented algebraic specification. Technical Report IS–RR–98–0017F, Japan Advanced Institute for Science and Technology, June 1998. Submitted for publication.
7. Răzvan Diaconescu and Kokichi Futatsugi. *CafeOBJ Report: The Language, Proof Techniques, and Methodologies for Object-Oriented Algebraic Specification*. World Scientific, 1998. AMAST Series in Computing, Volume 6.
8. Joseph Goguen. The logic of inexact concepts. *Synthese*, 19:325–373, 1968–69.
9. Joseph Goguen. Types as theories. In George Michael Reed, Andrew William Roscoe, and Ralph F. Wachter, editors, *Topology and Category Theory in Computer Science*, pages 357–390. Oxford, 1991. Proceedings of a Conference held at Oxford, June 1989.
10. Joseph Goguen. Towards a social, ethical theory of information. In Geoffrey Bowker, Leigh Star, William Turner, and Les Gasser, editors, *Social Science, Technical Systems and Cooperative Work: Beyond the Great Divide*, pages 27–56. Erlbaum, 1997.
11. Joseph Goguen. An introduction to algebraic semiotics, with applications to user interface design. In Chrystopher Nehaniv, editor, *Computation for Metaphors, Analogy and Agents*, pages 242–291. Springer, 1999. Lecture Notes in Artificial Intelligence, Volume 1562.
12. Joseph Goguen. Social and semiotic analyses for theorem prover user interface design. *Formal Aspects of Computing*, 11:272–301, 1999. Special issue on user interfaces for theorem provers.
13. Joseph Goguen and Rod Burstall. Institutions: Abstract model theory for specification and programming. *Journal of the Association for Computing Machinery*,

39(1):95–146, January 1992.

14. Joseph Goguen and Răzvan Diaconescu. Towards an algebraic semantics for the object paradigm. In Hartmut Ehrig and Fernando Orejas, editors, *Proceedings, Tenth Workshop on Abstract Data Types*, pages 1–29. Springer, 1994. Lecture Notes in Computer Science, Volume 785.
15. Joseph Goguen, Kai Lin, Akira Mori, Grigore Roşu, and Akiyoshi Sato. Distributed cooperative formal methods tools. In Michael Lowry, editor, *Proceedings, Automated Software Engineering*, pages 55–62. IEEE, 1997.
16. Joseph Goguen, Kai Lin, Akira Mori, Grigore Roşu, and Akiyoshi Sato. Tools for distributed cooperative design and validation. In *Proceedings, CafeOBJ Symposium*. Japan Advanced Institute for Science and Technology, 1998. Namazu, Japan, April 1998.
17. Joseph Goguen and Grant Malcolm. *Algebraic Semantics of Imperative Programs*. MIT Press, 1996.
18. Joseph Goguen and Grant Malcolm. Hidden coinduction: Behavioral correctness proofs for objects. *Mathematical Structures in Computer Science*, 9(3):287–319, June 1999.
19. Joseph Goguen and Grant Malcolm. A hidden agenda. *Theoretical Computer Science*, to appear. Also UCSD Dept. Computer Science & Eng. Technical Report CS97–538, May 1997.
20. Joseph Goguen, Grant Malcolm, and Tom Kemp. A hidden Herbrand theorem: Combining the object, logic and functional paradigms. In Catuscia Palamidessi, Hugh Glaser, and Karl Meinke, editors, *Principles of Declarative Programming*, pages 445–462. Springer Lecture Notes in Computer Science, Volume 1490, 1998. Full version to appear in *Electronic Journal of Functional and Logic Programming*, MIT, 1999.
21. Joseph Goguen and Grigore Roşu. Hiding more of hidden algebra. In Jeannette Wing, Jim Woodcock, and Jim Davies, editors, *FM'99 – Formal Methods*, pages 1704–1719. Springer, 1999. Lecture Notes in Computer Sciences, Volume 1709, Proceedings of World Congress on Formal Methods, Toulouse, France.
22. Joseph Goguen and Grigore Roşu. A protocol for distributed cooperative work. In Gheorghe Stefanescu, editor, *Proceedings, FCT'99, Workshop on Distributed Systems*, pages 1–22. Elsevier, 1999. (Iaşi, Romania). Also, Electronic Lecture Notes in Theoretical Computer Science, Elsevier Volume 28, to appear 1999.
23. Joseph Goguen, Timothy Winkler, José Meseguer, Kokichi Futatsugi, and Jean-Pierre Jouannaud. Introducing OBJ. In Joseph Goguen and Grant Malcolm, editors, *Software Engineering with OBJ: Algebraic Specification in Action*. Kluwer, 2000. Also Technical Report SRI-CSL-88-9, August 1988, SRI International.
24. Rolf Hennicker and Michel Bidoit. Observational logic. In *Algebraic Methodology and Software Technology (AMAST'98)*, volume 1548 of *Lecture Notes in Computer Science*, pages 263–277. Springer, 1999.
25. William Labov. The transformation of experience in narrative syntax. In *Language in the Inner City*, pages 354–396. University of Pennsylvania, 1972.
26. Charlotte Linde. The organization of discourse. In Timothy Shopen and Joseph M. Williams, editors, *Style and Variables in English*, pages 84–114. Winthrop, 1981.
27. Charlotte Linde. *Life Stories: the Creation of Coherence*. Oxford, 1993.

28. Eric Livingston. *The Ethnomethodology of Mathematics.* Routledge & Kegan Paul, 1987.
29. Nicholas Merriam and Michael Harrison. What is wrong with GUIs for theorem provers? In Yves Bartot, editor, *Proceedings, User Interfaces for Theorem Provers*, pages 67–74. INRIA, 1997. Sophia Antipolis, 1–2 September 1997.
30. George A. Miller. The magic number seven, plus or minus two: Some limits on our capacity for processing information. *Psychological Science*, 63:81–97, 1956.
31. Peter Padawitz. Swinging types = functions + relations + transition systems, 1999. `issan.informatik.uni-dortmund.de/~peter`. Submitted to *Theoretical Computer Science.*
32. Jenny Preece, Yovonne Rogers, et al. *Human-Computer Interaction.* Addison Wesley, 1994.
33. Grigore Roşu. Behavioral coinductive rewriting. In Kokichi Futatsugi, Joseph Goguen, and José Meseguer, editors, *OBJ/CafeOBJ/Maude at Formal Methods '99*, pages 179–196. Theta (Bucharest), 1999. Proceedings of a workshop in Toulouse, 20 and 22 September 1999.
34. Grigore Roşu and Joseph Goguen. Circular coinduction, 1999. UCSD Technical Report; revised version submitted for publication.
35. Grigore Roşu and Joseph Goguen. Hidden congruent deduction. In Ricardo Caferra and Gernot Salzer, editors, *Automated Deduction in Classical and Non-Classical Logics*, pages 252–267. Springer, 2000. Lecture Notes in Artificial Intelligence, Volume 1761; papers from a conference held in Vienna, November 1998.
36. David E. Rumelhart and J.L. McClelland, editors. *Parallel Distributed Processing.* MIT, 1986.
37. Ferdinand de Saussure. *Course in General Linguistics.* Duckworth, 1976. Translated by Roy Harris.
38. Ben Shneiderman. *Designing the User Interface.* Addison Wesley, 1997.
39. SSL 3.0 specification. `home.netscape.com/eng/ssl3/ssl-toc.html`.
40. XML™ (Extensible Markup Language) specification. `www.w3.org/XML/`.
41. XSL (Extensible Stylesheet Language) specification. `www.w3.org/TR/WD-xsl`.

A. Sample DUCK Code

A DUCK text for proving correctness of the traditional array-with-pointer implementation of stack is given below. It divides into two parts, the first for proofs and the second for display. In addition to the main theorem, there are three lemmas. Each lemma is proved by induction, while the main goal is proved by coinduction. Each proof part begins with the keyword "`proof:`" and ends with "`[]`", which signals that KUMO should try to finish the proof using its built in tactics that require no user intervention, including reduction, the elimination rules for universal quantifiers, conjunction, and so on. A name may be optionally given after the keyword "`proof:`". Comment lines begin with "`***`". More detail on this proof can be obtained from its homepage, which has the URL `www.cs.ucsd.edu/groups/tatami/demos/array/`, and information about the latest version of Kumo can be obtained from the URL `www.cs.ucsd.edu/groups/tatami/kumo/`.

```
project: Stack
```

```
getspec: PTR+ARR.1.klin
relation:
   op _R_ : Stack Stack -> Bool .
   var I1 I2 : Nat .
   var A1 A2 : Arr .
   eq (s I1 || A1) R (s I2 || A2) =
          I2 == I1 and A1[I1] == A2[I1] and (I1 || A1) R (I1 || A2) .
   eq (0 || A1) R (0 || A2) = true .
[]
cobasis: {top, pop}
*****************************************************
proof: <<Lemma1>>
    goal: (forall X  Y : Nat)
            ((s Y) <= X  implies (exists Z : Nat)(X = s Z  and  Y <= Z )) .
by: induction on X with scheme {0,s} []
*****************************************************
proof: <<Lemma2>>
    goal: (forall X  Y : Nat) ( s X <= Y  implies  X <= Y  ) .
by: induction on X with scheme {0,s} []
*****************************************************
proof: <<PutAndBar>>
    goal: (forall I J N : Nat  A : Arr)
             (  I <= J   implies  (I || put(N,J,A)) R (I || A) ) .
by: induction on I with scheme {0,s} []
*****************************************************
proof: <<StackThm>>
  goal:  pop empty = empty  and
   ((forall N I : Nat  A : Arr)  top push(N, I || A) = N  ) and
   ((forall N I : Nat  A : Arr)  pop push(N, I || A) = (I || A) ) .
by: coinduction []
*****************************************************
display:
   title: "Lemma for Pointer Array Implementation of Stack"
<<Lemma1>>
   subtitle: "We prove a basic property of the order on natural numbers."
[]
*****************************************************
display:
   title: "Lemma for Pointer Array Implementation of Stack"
<<Lemma2>>
   subtitle: "We prove a basic property of the order on natural numbers."
[]
*****************************************************
display:
   title: "Key Lemma for Stack Implementation"
   gethomepage:  /home/klin/he/putandbarhome.html
<<PutAndBar>>
   subtitle: "We prove the key lemma for stack implementation."
[]
*****************************************************
display:
   title: "Pointer Array Implementation of Stack "
   gethomepage: /home/klin/he/stackhome.html
   getspecexpl: /home/klin/he/stackspecexp.html
```

```
<<StackThm>>
   subtitle: "We show that array-with-pointer implementation of stack is correct."
   getexpl:  /home/klin/he/stackexp1.html
   <<StackThm.1>>
   subtitle: "We prove <math>R</math> is <delta/>-congruence."
   getexpl:  /home/klin/he/stackexp2.html
   <<StackThm.2>>
   subtitle: "We prove the main result."
   getexpl:  /home/klin/he/stackexp4.html
[]
```

B. Sample XSL Code

The XML files produced by KUMO as described in Section 2.1 are used together with an XSL style file to generate the HTML that is actually displayed by the user's browser. Below is the XSL code for the output of a conjunction elimination rule application:

```
<xsl:template match="kumo.logic.fol.ConjunctionEli">
   <a href="{constant(conjel)}" target="back">Conjunction elimination</a>
   yields the following <xsl:apply-templates match="count"/> subgoals:
</xsl:template>
```

It says that the generated HTML will contain the text "`Conjunction elimination yields the following` K `subgoals:`" where the integer K is number of subgoals, obtained by calling another XSL rule, named "`count`"; a link to the conjunction elimination tutorial page, the URL of which is named by the constant "`conjel`", is also inserted. (The list of subgoals is already present in the XML tree that is being processed and therefore does not need to be inserted by this rule.)

C. A Formal Description of 2-Doags

Here we briefly define the data structure that is used for storing validations. If N is a set, then N^* denotes the set of finite lists over N. A **2-doag** D consists of a set N of **nodes**, a set F of **fans**, two functions $d_0 : F \to N$ and $d_1 : F \to N^*$ (called **source** and **target**), a set W of "labels" plus a labeling function $l : N \to W$, such that

1. for each node n, the fans with source n are ordered;
2. for each fan f, the target nodes of f are ordered;
3. let $D^\flat$ be the ordinary directed ordered graph constructed from D to have the same nodes N as D, and to have as its edges from n to n' pairs (f, n') such that n' is a target node of a fan f with source n, with edges ordered by the ordering of fans plus that of edges within fans; then $D^\flat$ must be acyclic.

The labeling function is just to accommodate the additional information attached to nodes mentioned in Section 2.2. Note that $D^\flat$ need not have a unique root and need not be connected. 2-doags are really a kind of hypergraph, and fans are really hyperedges, but we use the "2-dimensional" language because of its suggestiveness for our application. A 2-occurrence in D is just a path in $D^\flat$.

Chapter 4

Proof Assistance for Equational Specifications Based on Proof Obligations*

Masaki Ishiguro [a] and Ataru T. Nakagawa [b]

[a]Information Technologies Development Dept.
Mitsubishi Research Institute, Inc.
3-6, Otemachi 2-Chome, Chiyda-ku, Tokyo, Japan
masa@mri.co.jp

[b]Software Engineering Laboratory
SRA, Inc.
12, Yotsuya 3-Chome, Shinjyuku-ku, Tokyo, Japan
nakagawa@sra.co.jp

In this paper we consider the equational fragment of CafeOBJ, called eCafeOBJ, and describe a mechanism for checking correctness of specifications based on proof obligations associated with model-theoretic assertions.

In (full) CafeOBJ, such declarations as *views*, *imports*, and *constructors* restrict models as if additional axioms or implications, which are not in general expressible in CafeOBJ itself, were given. It behooves us to convince ourselves that these extra assertions are indeed valid consequences of the explicitly given axioms. Thus we get proof obligations, in an informal sense, whenever those language constructs are used.

If we narrow our focus to eCafeOBJ, (suffucient conditions of) some proof obligations are expressible in eCafeOBJ. In such cases, we may generate the proof obligations in an equational form purely syntactically. Once reduced to such a form, standard provers for eCafeOBJ sentences and standard proof techniques based on term rewriting engines can be applied. In addition, those generated obligations can be used deliberately to check correctness of specifications with the help of user-defined predicates, to check sufficient completeness, and so on. To put the idea into practice, we implemented a proof obligation generator as part of a broader proof assistance system for full CafeOBJ.

After introducing the general idea and giving little demonstrations that illustrate the idea, we present an example in which we show that a proof of a property of a parameterised module automatically leads to a proof of the property of any instantiated module, so long as the proof obligations of the view declaration are discharged.

*This research has been funded in part by the Advanced Software Enrichment Project of the Information-Technology Promotion Agency, Japan(IPA).

1. Introduction

In CafeOBJ[8], such declarations as *views*, *imports*, and *constructors* assert model-theoretic properties. Though semantics of these declarations are well-defined in theory[7], it is difficult in general to provide a mechanical support for verifying such assertions, even if we assume heavy interactions with the user. The first hurdle is to find a suitable language in which all those assertions are expressible. It is no wonder that the current CafeOBJ processor treats these declarations only syntactically and ignores implicit assertions.

If we restrict ourselves to the equational fragment of CafeOBJ, called eCafeOBJ in the sequel, this difficult problem becomes a little more manageable. In some happy cases we can reduce (sufficient conditions of) the assertions associated with these declarations to equations or sort (type) predicates of eCafeOBJ, and apply provers and proof techniques to them in the framework of equational logic[1]. In this paper all the formulations are devised to utilise the term rewriting facilities of the CafeOBJ system[16], but there are many possibilities as to the choice of existing tools once translation schemes are established.

The equational formulations and the algorighms used to generate equational formulae are more or less standard. Our contribution in this regard is to have put existing folklore ideas into practice, and to have implemented a practical mechanism. The emphasis is on ease of use rather than comprehensiveness. We do not claim that our mechanism deals with an adequate class of equations, whatever definition is used for "adequateness". Some devises are applicable only to initial models. The generated equations may not be provable by the facilities of the CafeOBJ system, even if the original assertions are valid.

On the methodological side, we considered the possibility to prove user-supplied theorems by the above mechanism. The basic idea is that, if an equation is a theorem of the set of axioms in a given module, (a) the equation can be embedded into a module who protectively imports the original module, or (b) there is a (correct) view to the original module from the module with the same signature but with only the equation as an axiom. Thus the problem of theoremhood may be stated as a problem of proof obligations in the syntax of eCafeOBJ. We illustrate this usage of our mechanism by a simple example.

As a promising way to employ our mechanism, we also show an example of a proof involving a parameterised module. As a norm[11], parameterised modules support reuse of codes, and offer the possibility of reusing proofs, and/or of replacing the need for thousands of proofs by a single one. The example in the paper, borrowed from [10], is a demonstration of how this observation taks a concrete form when our mechanism is applied to eCafeOBJ codes.

We developed a systematic proof support system for (full) CafeOBJ specifications, integrating the obligation generation facilities explained in this paper with the inductive theorem prover[5] implemented on top of the term rewriting engine Maude[14]. The system also generates command sequences for the CafeOBJ system, to prove equations or sort predicates; in addition, experiments have been going on to use SPIKE system[3,1]. We also introduced *Forsdonnet* which extends HTML for handling formal specifications distributed over a network, providing the above proof mechanism via Netscape and Emacs interface[9,13,17].

The paper is organised as follows. After an overview in Section 2, we introduce the basic

[1]In some cases membership logic[4,15] is also used.

idea in Section 3 with examples. In Section 4 user-supplied theorems are considered within our framework. Section 5 explains in some detail an example of proofs for parameterised modules. Section 6 concludes the paper with a couple of remarks.

Throughout the paper, we assume some familiarity with equational specifications in general, and eCafeOBJ in particular.

2. Overview

Some eCafeOBJ declarations impose restrictions on models, although they do not look like axioms. Among these declarations, we focus on *views*, *imports*, and *constructors* in this paper.

A *view* declaration defines signature morphisms that preserve the satisfaction relation. The axioms of the source module must hold, under the derived translation, in the target module.

A module *import* states that the imported module is embedded into the importing module so that all the declarations of the former are available in the latter; in addition, depending on the *import mode*, a model of the latter has to have a certain counterpart of the former.

Constructor declarations on a set of operators of the same coarity assert that those operators constitute a (possibly non-free) constructor set.

In treating these declarations, the CafeOBJ system takes a mostly syntactical point of view, and does nothing to check whether these assertions are valid. To ignore the assertions at the level of a general CafeOBJ processor is a good decision. Even resitricted to the equational fragment, such assertions are not easy to formulate in a formal language, let alone in CafeOBJ itself. Even if they are so formulated, there is no decision procedure in general. Even if for a restricted class of specifications there is such a decision procedure, unless a very clever proof support is given, it is not realistic nor worthwhile to try to prove every assertion for every specification.

In the practice of eCafeOBJ, in the absence of a mechanical support from the CafeOBJ system, the user has tried to ensure by manual reasoning that the import modes he is using are correct ones, that he has written adequate axioms for non-constructors, that the views are indeed specification morphisms, and so on. In many simple cases such assurances are easily obtainable and do not impose too heavy a burden on the part of the user. At times, however, he is uncertain about the correctness of such a declaration, and wants to construct a detailed proof thereof. The argument used there is probably partly formal, but the overall proof is an informal one. And the formal parts themselves are mostly ad hoc.

While this way of proving non-trivial theorems fits the practice of working mathematicians[12], a systematic formal proof system would be helpful in many situations. Especially, the boring part of proofs, such as enumerating all the case distinctions, is certainly worth mechanising, to save the intellectual efforts of the user for more creative tasks. We intended to implement such a system. We also intended to provide proof tools, base on the CafeOBJ system and Maude, to prove theorems automatically. The major aim of the latter tools is limited in scope: they are to give a little assistance here and there for the user to construct a large and difficult proof.

Under some conditions, the assertions by *views*, *imports*, and *constructors* in eCafeOBJ can be reduced to eCafeOBJ equations or sort predicates, or sufficient conditions for these assertions can be extracted in such forms. By a slight abuse of terminology, in the sequel we may call such equations and sort predicates as "the" proof obligations of the declarations. Representing proof obligations in equational sentences not only helps the user understand the specifications by clarifying what he is actually writing, but also makes it possible to treat and prove them by equational reasoning, and/or inductive proofs if the initial semantics is involved. An advantages of having such equations in the syntax of eCafeOBJ and using proof mechanisms within the CafeOBJ system is that the user does not need to know the gory details of additional languages nor provers.

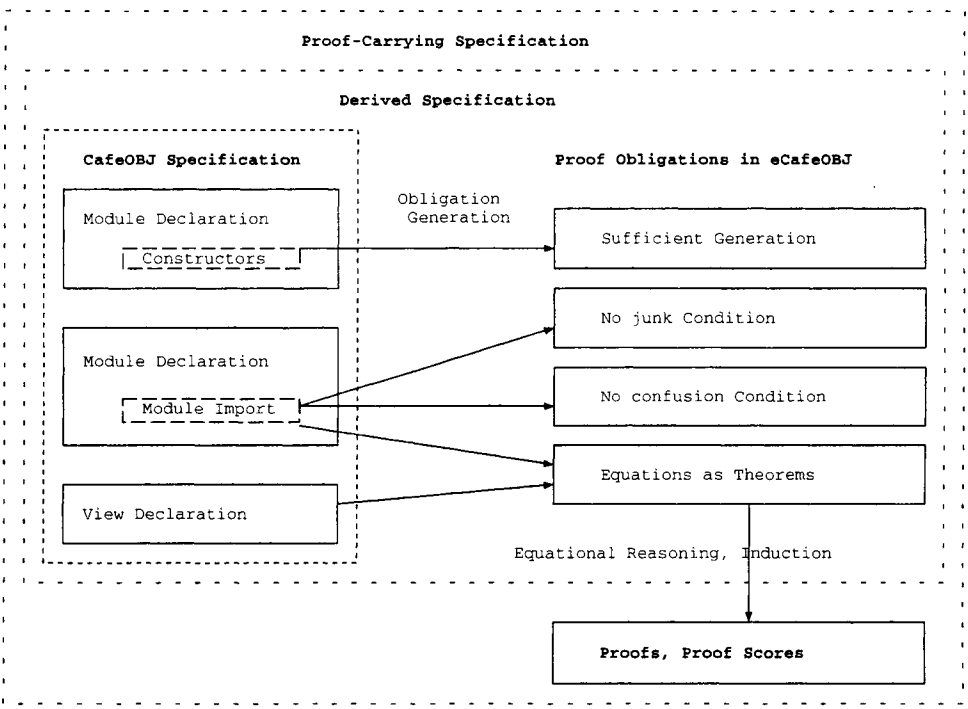


Figure 4.1. An overview of the mechanism

Figure 4.1 illustrates how our mechanism is to be used in practice. The user writes a plain eCafeOBJ specification, possibly including such declarations as views. The specification is a concise and (hopefully) exact representation of the user's intention. The proof obligation generation tool expands "implicit" assertions in the the specification into equational sentences or sort predicates (called proof obligations in eCafeOBJ sen-

tences in the figure). These assertions can be proved by the equational reasoning or induction. Together with the proofs of the generated proof obligations of the original specification, those formal codes can be considered as a verified/assured specification, or a proof-carrying specification.

3. Extracting Proof Obligations

We concentrate on three kinds of declarations in CafeOBJ specification, which we consider as important components for constructing specifications and proofs. For each kind, we first give a concise description of associated proof obligations (for detail, see [8]) and then provide the detail of the obligation extraction mechanism.

3.1. Proof Obligations of Constructors

We consider here only the initial semantics and modules without parameters.

Constructors provide a handy mechanism to create concise inductive proofs and case analyses, and give justifications to test-set inductions in the manner of [2]. The assertions by constructor declarations are related to the notion of *sufficient generation* ([6]):

Sufficient Generation

A specification $\mathrm{SP} = (\Sigma, \Gamma)$ is sufficiently generated by a subspecification[2] $\mathrm{SP}_c = (\Sigma_c, \Gamma_c)$ iff for every Σ-ground term t of sort s, there is a Σ_c-ground term t_c of the same sort such that t provably equals t_c under Γ.

In general, a specification in eCafeOBJ need not have a constructor declaration at all, nor is it required that constructor declarations be systematically given for all the sorts. Thus the above property is in general too strong. But under the condition that every sort has an associated set of constructors[3], by considering the subspecification defined thereby, we can generate the proof obligations of constructor declarations, by dint of [5], as follows.

Let $\mathrm{M} = (\Sigma, \Gamma)$ be a (flat) module, where Σ is the signature and Γ is the set of equations. Further, let $\mathcal{C}$ be the set of operators declared as constructors, and $\mathcal{D}$ be the set of other operators. Then a new module M' is constructed by modyfing M as follows.

1. For each connected component of sorts in Σ, declare a fresh sort, and make it a supersort of the whole component.

2. Replace each $f : s_1 \dots s_n \to s$ in $\mathcal{D}$ with an operator $f : c(s_1) \dots c(s_n) \to c(s)$, where $c(s)$ denotes the fresh sort introduced above as a supersort of s, and similarly for $c(s_i)$'s.

Then the proof obligations are, for each $f : s_1 \cdots s_n \to s \in \mathcal{D}$,

$$\forall \overline{x : s}.\ (f(\overline{x}))\ :\mathrm{is}\ s$$

[2]This means Σ_c is a subsignature of Σ, $\Gamma_c \subseteq \Gamma$, and Γ_c consists of Σ_c-sentences.

[3]Note that by definition any constructor of a sort is also a constructor of its supersort.

where $\overline{x}$ the list of distinct variables $x_1, \ldots, x_n$ and $\forall \overline{x : s}$ abbreviates $\forall x_1 : s_1 \ldots \forall x_n : s_n$. Note that ":is" in the formula is the sort (membership) predicate of eCafeOBJ: it evaluates to true if the least sort of the term is less than or equal to s. A formal justification of this scheme can be found in [4].

For a simple example, consider the natural numbers under addition defined as follows.

```
module! NAT {
  [ Nat ]
  op 0 : -> Nat { constr }
  op s_ : Nat -> Nat { constr }
  op _+_ : Nat Nat -> Nat
  vars M N : Nat
  eq M + 0 = M .
  eq M + s N = s(M + N) .
}
```

From this module, our system generates the following module[4].

```
module! NAT' {
  [ Nat < Nat+ ]
  op 0 : -> Nat { constr }
  op s_ : Nat -> Nat { constr }
  op _+_ : Nat+ Nat+ -> Nat+
  vars M N : Nat
  eq M + 0 = M .
  eq M + s N = s(M + N) .
}
```

The system also generates the proof obligation below.

```
N:Nat + M:Nat :is Nat
```

Depending on the user's direction, the system then generates an induction or case-analysis scheme in the form of reduction commands of the CafeOBJ system, or an input to the inductive theorem prover developed on top of Maude. If a case-analysis scheme for the CafeOBJ system is chosen, our system generates the following sequence of commands.

```
open NAT' .
op n : -> Nat .
reduce n + 0 :is Nat .
op m : -> Nat .
reduce n + s(m) :is Nat .
close
```

[4]The generated names are actually more criptic.

3.2. Proof Obligations of Imports

The assumption here is that imported modules have the initial semantics, and have canonical (or free) constructors.

Module imports of eCafeOBJ confer a modular structure on specifications. In addition, they restrict admissible models of importing modules, using the concept *import mode*. There are three import modes in eCafeOBJ, called *protecting*, *extending*, and *using*.

- If the mode is *protecting*, the reduct of any model of the importing module must be a model of the imported one.
- If it is *extending*, the reduct of any model of the importing module must have a submodel that is also a model of the imported one.
- If it is *using*, no restriction.

In terms of "no junk" and "no confusion", therefore, *protecting* requires both conditions, *extending* requires only "no confusion", and *using* requires neither. Our system formulates proof obligations based on these conditions. Let M be a module with a canonical constructor set $\mathcal{C}$, and let M' import M. Then "no confusion" is assured by proving, in M', the following assertions, where $f_i : s_1 \ldots s_n \to s$, $f_j : s'_1 \ldots s'_m \to s'$ range over $\mathcal{C}$ and $\overline{x}$ etc. are abbreviations as before.

$$\forall \overline{x : s}\ \forall \overline{x' : s'}.\ f_i(\overline{x}) \neq f_j(\overline{x'}) \ \text{ if } \ f_i \neq f_j$$

$$\forall \overline{x : s}\ \forall \overline{x' : s}.\ f_i(\overline{x}) = f_i(\overline{x'}) \Rightarrow x_i = x'_i \ \text{ for each } \ 1 \leq i \leq n$$

For a technical reason the latter kind of assertions is easier to deal with in the contrapositive form, as

$$\forall \overline{x : s}\ \forall \overline{x' : s}.\ x_i \neq x'_i \Rightarrow f_i(\overline{x}) \neq f_i(\overline{x'})$$

The inequalities cannot be handled directly, and the iequality predicate of eCafeOBJ is used instead. For example, consider the following module that imports the previous `NAT` module:

```
module! NAT-EX {
  extending (NAT)
  op _*_ : Nat Nat -> Nat
  vars N M : Nat
  eq N * 0 = 0 .
  eq N * s(M) = N * M + N .
}
```

Assuming `0`, `s` constitute a canonical constructor set, the proof obligations of this import are

```
  0 =/= s(N:Nat) = true
  s(N:Nat) =/= s(M:Nat) = true if N =/= M
```

At the moment we cannot go further. The problem here is to ensure the nonexistence of such N and M, and we do not have a tool suitable for such a purpose. The concept of abstract model checking may be worth considering in this respect.

Turning to "no junk" condition, the scheme adopted by our system is quite similar to the case of *constructor* declarations explained in Section 3.1. For example, if we replace `extending` with `protecting` in `NAT-EX`, the system generates a module

```
module! NAT-EX' {
  [ Nat < Nat+ ]
  op 0 : -> Nat { constr }
  op s_ : Nat -> Nat { constr }
  op _+_ : Nat Nat -> Nat
  op _*_ : Nat+ Nat+ -> Nat+
  vars M N : Nat
  eq M + 0 = M .
  eq M + s N = s(M + N) .
  eq N * 0 = 0 .
  eq N * s(M) = N * M + N .
}
```

and the proof obligation

```
N:Nat * M:Nat :is Nat
```

We may use inductive proofs in this example. The CafeOBJ command sequence for this obligation is as follows.

```
open NAT-EX' .
op n : -> Nat .
** base case
reduce n * 0 :is Nat .
** induction step
op m : -> Nat .
eq n * m :is Nat = true .
reduce n * s(m) :is Nat .
close
```

As a special case, if the importing module introduces no new sort nor operator, the system extracts the axioms in the module as the proof obligations. One reason for distinguishing this special case is that theorems written by the user may be regarded as proof obligations of imports. See Section 4 for a further explanation.

3.3. Proof Obligations of Views

Views provide an instantiation mechanism for parameterised modules. Instantiation is achieved by a view declaration that defines a specification morphism, i.e. a signature morphism that preserves the satisfaction relation. By discharging the proof obligations of

views, the user can not only abstract a parameterised specification out of detail implementations of the actual parameter specifications, but also prove properties of instantiated modules before instantiation.

Given modules $M = (\Sigma, \Gamma)$ $M' = (\Sigma', \Gamma')$, a view from M to M' is a signature morphism $\phi : \Sigma \to \Sigma'$ such that

$$\Gamma' \models \phi^{\#}(e) \text{ for each } e \in \Gamma$$

where $\phi^{\#}$ is the translation of terms induced by ϕ. The proof obligations of views are exactly such $\phi^{\#}(e)$'s.

Depending on the user's request, the system generates an input either to the CafeOBJ system, or to the inductive theorem prover that runs on Maude. In the case when M' has the initial semantics, the system may generate an induction scheme for the CafeOBJ system. As an example, given a module

```
module* ASSOC {
  [ S ]
  op _*_ : S S -> S
  vars X Y Z : S
  eq (X * Y) * Z = X * (Y * Z) .
}
```

and a view

```
view +is-assoc from ASSOC to NAT {
  sort S -> Nat
  op (_*_) -> (_+_)
}
```

the proof obligation is

```
  (N:Nat + M:Nat) + L:Nat = N + (M + L)
```

and the CafeOBJ command sequence using induction on the last variable is

```
  open NAT .
  ops n m : -> Nat .
  ** base case
  reduce (n + m) + 0 == n + (m + 0) .
  ** induction step
  op l : -> Nat .
  eq (n + m) + l = n + (m + l) .
  reduce (n + m) + s(l) == n + (m + s(l)) .
  close
```

4. Using Proof Obligations for Arbitrary Equations

Suppose the user wants to prove the (right) distributive law in the module **NAT-EX**. The law can be stated as a plain sentence in eCafeOBJ, as

```
(N:Nat + M:Nat) * L:Nat = N * L + M * L
```

To prove this law is not hard, and we may use the CafeOBJ system or the Maude-based inductive theorem prover, with the help of an induction scheme. Before that, however, there is a little problem. How can we state that this equation is a theorem of **EX-NAT**? One possibility is to design a small language that may express the idea of axioms, theories, theoremhood, satisfaction, and so on. *Forsdonnet* is one such language. We here consider another possibility: to use eCafeOBJ itself to express theoremhood. For the purpose, the user builds a module and a view as follows.

```
module DIST-R {
  [ S ]
  op _+_ : S S -> S
  op _*_ : S S -> S
  vars X Y Z : S
  eq (X + Y) * Z = X * Z + Y * Z .
}

view +*-is-r-distributive from DIST-R to NAT-EX {
  sort S -> Nat
  op _+_ -> _+_
  op _*_ -> _*_
}
```

Then the proof obligation of the view is exactly the theorem the user wants to state and prove. This method looks a roundabout way of arranging things, but has a couple of advantages.

- The user may work entirely within the language of eCafeOBJ.
- The tools developed for proof obligations of views can be used without any modification.

There is another possible way to state the original problem within the language of eCafeOBJ. If the user writes a module

```
module! DIST-+* {
  extending (NAT-EX)
  vars N M L : Nat
  eq (N + M) * L = N * L + M * L .
}
```

then the "no confusion" condition states that the equation does not newly identify any two distinct elements, which means that the equation already holds in NAT-EX, i.e., that the equation is a theorem there. As a special case of imports, when no new sort nor operator is introduced, the system simply extracts axioms as the proof obligations. Unlike the general case, such obligations can often be given easy proofs. In this example, a structural induction is a good possibility.

5. Proof Construction for Parameterised Modules

With regard to proofs, the module parameterisation mechanism in eCafeOBJ provides a powerful abstraction facility. As an application of our system, we here show a systematic example that exploits this facility. The following obvious observation forms the basis of this example.

Fact

> Let M[P] be a module with parameter P, M' be another module, V be a view from P to M', and M[V] be the instantiation of M[P] along V.
> If an eCafeOBJ sentence e is proven in M[P] and the proof obligations of V are discharged, the translation of e induced by V is provable in M[V].

The fact holds since in M[V] all the axioms of M[P] are already there or can be proven as proof obligations of V.

The subject in question is to sort lists, and is an adaptation of an example in [10]. A specification QSORT, which defines the quicksort algorithm, is to be defined as a module where list elements are parameterised by a module TOSET, which represents an arbitrary totally ordered set. We consider two instantiations of this parameterised module: one is by the module NAT< of natural numbers and the other is by the module LEX-ORDER[NAT] of the lexicographical ordering of lists of natural numbers. Relations among those modules are described in Figure 4.2. We begin by showing a basic specification of lists.

```
module! LIST (X :: TRIV) {
  protecting (NAT)
  [ NeList < List ]
  op nil : -> List  { constr }
  op cons : Elt List -> NeList  { constr }
  op length : List -> Nat
  op _in_ : Elt List -> Bool
  op del1 : Elt List -> List
  op append : List List -> List
  vars X Y : Elt
  vars L K : List
  eq length(nil) = 0 .
  eq length(cons(X, L)) = s(length(L)) .
  eq del1(X, nil) = nil .
  eq del1(X, cons(X, L)) = L .
  cq del1(X, cons(Y, L)) = cons(Y, del1(X, L)) if X =/= Y .
```

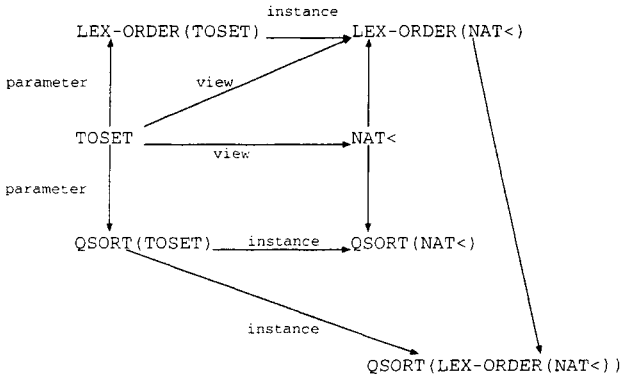


Figure 4.2. Relations among theories, parametric modules and instance modules

```
  eq X in nil = false .
  eq X in cons(X, L) = true .
  cq X in cons(Y, L) = X in L if X =/= Y .
  eq append(L, nil) = L .
  eq append(nil, L) = L .
  eq append(cons(X, L), K) = cons(X, append(L, K)) .
}
```

5.1. Checking User-Defined Predicates

Now we define predicates `sorted` and `perm`. The former returns true iff its argument is a sorted list, while the latter returns true iff the second argument permutes the first. The ordering is defined in a module `TOSET`, which we show first.

```
module* TOSET {
  [ Elt ]
  op _<_ : Elt Elt -> Bool
  vars E1 E2 E3 : Elt
  eq E1 < E1 = false .
  cq E1 < E3 = true if (E1 < E2) and (E2 < E3) .
  cq (E1 < E2) or (E2 < E1) = true if E1 =/= E2 .
}

module! SORTED-LIST (X :: TOSET) {
  protecting (LIST(X <= view to X { sort Elt -> Elt }))
  op sorted : List -> Bool
  op perm : List List -> Bool
  vars X Y Z W : Elt
```

```
  vars L K : List
  eq sorted(nil) = true .
  eq sorted(cons(X, nil)) = true .
  eq sorted(cons(X, cons(Y, L))) =
     (X < Y or X == Y) and sorted(cons(Y, L)) .
  eq perm(nil, nil) = true .
  eq perm(nil, cons(X, L)) = false .
  eq perm(cons(X, L), nil) = false .
  eq perm(cons(X, nil), cons(X, nil)) = true .
  cq perm(cons(X, nil), cons(Y, nil)) = false if X =/= Y .
  eq perm(cons(X, nil), cons(Y, cons(Z, L))) = false .

  eq perm(cons(X, cons(Y, L)), cons(Z, nil)) = false .
  cq perm(cons(X, cons(Y, L)), cons(Z, cons(W, K)))
      = perm(cons(Y, L), del1(X, cons(Z, cons(W, K))))
      if X in cons(Z, cons(W, K)) .
  cq perm(cons(X, cons(Y, L)), cons(Z, cons(W, K)))
      = false if not (X in cons(Z, cons(W, K))) .
}
```

The proof obligations of the *protecting* import concern only the Boolean sort `Bool`, and correspond to the intuitive correctness of the definitions of the predicates, as follows.

- The "no junk" condition means, among other things, that any term of the form `sorted(t)` or `perm(t,t')` equals either `true` or `false`. This means these predicates are totally defined.
- The "no confusion" condition means that `true` = `false` does not hold. Coupled with the above condition, this means that any term of the form `sorted(t)` or `perm(t,t')` equals exactly one of `true` or `false`. Thus these predicates are well-defined.

According to Section 3.2, the "no junk" part of the obligations consists of two sort predicates in the generated module `SORTED-LIST'`, as

```
sorted(L) :is Bool
perm(L,L') :is Bool
```

The "no confusion" part is simply

```
true =/= false = true
```

5.2. Proofs in Parameterised Modules

Our main task is to create an abstract proof for the operation `qsort` that sorts lists of any totally ordered set.

```
module! QSORT (TS :: TOSET) {
  protecting (SORTED-LIST(X <= view to TS{ sort Elt -> Elt }))
```

```
  ops smaller larger : Elt List -> List -- { strat: (1 2 0) }
  op qsort : List -> List
  vars X Y : Elt
  var L : List
  eq smaller(X, nil) = nil .
  cq smaller(X, cons(Y, L)) = cons(Y, smaller(X, L)) if Y < X .
  cq smaller(X, cons(Y, L)) = smaller(X, L) if not (Y < X) .
  eq larger(X, nil) = nil .
  cq larger(X, cons(Y, L)) = cons(Y, larger(X, L)) if not (Y < X) .
  cq larger(X, cons(Y, L)) = larger(X, L) if Y < X .
  eq qsort(nil) = nil .
  eq qsort(cons(X, L)) = append(qsort(smaller(X, L)),
                                cons(X, qsort(larger(X, L)))) .
}
```

As usual, we say that a sorting algorithm is correct iff the result is a sorted list and is a permutation of the argument. In our setting this statement takes the form of two equations shown below.

```
sorted(qsort(L:List)) = true
perm(L:List, qsort(L)) = true
```

Since QSORT has the initial (free) semantics, those equations can be proved by inductive methods as well as a pure equational deduction. Cafe environment provides those inference tools and they are discussed in detail in other literatures [8,5]. On this example we also experimented with SPIKE.

5.3. Proof Abstraction and Views

We turn to checking proof obligations of two views. The first one has as target a natural number module.

```
module! NAT< {
  [ Nat ]
  op 0 : -> Nat  { constr }
  op s : Nat -> Nat  { constr }
  op _+_ : Nat Nat -> Nat
  op _<_ : Nat Nat -> Bool
  vars N N' : Nat
  eq 0 + N = N .
  eq s(N) + N' = s(N + N') .
  eq N < 0 = false .
  eq 0 < s(N) = true .
  eq s(N) < s(N') = N < N' .
}
```

Instantiation of QSORT by NAT< can be achieved by the following view and instantiation declarations.

```
view NAT-AS-TOSET from TOSET to NAT< {
  sort Elt -> Nat
  op _<_ -> _<_
}
module! NAT-QSORT {
  protecting (QSORT(NAT-AS-TOSET))
}
```

The proof obligations of this view are the following conditional equations:

```
 N1:Nat < N1 = false
 N1:Nat < N3:Nat = true if (N1 < N2:Nat) and (N2 < N3)
(N1:Nat < N2:Nat) or (N2 < N1) = true if N1 =/= N2
```

If we can prove these goals, we can safely conclude without any further proof that the correctness the quicksort algorithm, proved in the parameterised module QSORT, is also provable in the instantiated NAT-QSORT.

Next we show another case of instantiation which shows the usefulness of abstraction via the modules TOSET and LIST. This example uses the set of lists under lexicographical ordering as an instance of totally ordered sets. A specification of lexicographical ordering is given by the following module.

```
module! LEX-ORDER (TS :: TOSET) {
  protecting (LIST(X <= view to TS {sort Elt -> Elt}))
  op _<_ : List List -> Bool
  vars L L1 L2 : List
  vars E E1 E2 : Elt
  eq L < nil = false .
  eq nil < cons(E, L) = true .
  eq cons(E, L1) < cons(E, L2) = L1 < L2 .
  ceq cons(E1, L1) < cons(E2, L2) = true if E1 < E2 .
  ceq cons(E1, L1) < cons(E2, L2) = false if E2 < E1 .
}
```

Here the element set to be sorted is that of lists of a totally ordered set, where the list definition is imported from LIST. Instantiating the module QSORT by the lexicographical ordering of lists of a totally ordered set (natural numbers in this case) is done by the following declarations.

```
module! LEX-ORDER-NAT-LIST {
  protecting (LEX-ORDER(TS <= view to NAT< {
    sort Elt -> Nat
    op _<_ -> _<_  }))
}
view LEX-ORDER-AS-TOSET from TOSET to LEX-ORDER-NAT-LIST {
  sort Elt -> List
```

```
  op _<_ -> _<_
}
module! LEX-QSORT {
  protecting (QSORT(LEX-ORDER-AS-TOSET))
}
```

The proof obligations of this view declarations are the following three conditional equations in the module LEX-ORDER-NAT-LIST and all of them can be solved by inductions and by exhaustive case analyses.

```
L1:List < L1 = false
L1:List < L3:List = true if (L1 < L2:List) and (L2 < L3)
(L1:List < L2:List) or (L2 < L1) = true if L1 =/= L2
```

Continuing along this line, we can deduce properties of a sorting operation in other instantiated modules only by proving the proof obligations of the views used for instantiation. Thus we can say that the proofs for a parameterised module were abstracted away from the implementation of instantiations, and are reused after each instantiation. This proof method was achieved by taking advantage of parameterisation/instantiation mechanism already in eCafeOBJ, enhanced by a proof obligation generator.

6. Conclusions and Future Works

We explained that some of the eCafeOBJ declarations can be regarded as expressive assertions on models of specifications, and that in some cases such assertions are put in the form of eCafeOBJ sentences. Such a translation makes the user understand his specifications better, and opens the possibility of using various provers within the framework of eCafeOBJ.

We also provided an example which illustrates the usefulness of our approach. In the example, the proofs for parameterised modules were shown to be reusable without modification; for each instantiation, only the proof obligations of the view need to be proven anew.

We implemented a system based on the idea of proof obligation extractions, and made some experiments. The work is still at a very early stage, and there are many issues for future consideration. Firstly, the proof obligations the system generates are based on various assumptions. Of those assumptions, the presence of constructors is a little controvertial, since the current practice of eCafeOBJ does not systematically use constructor declarations. To ease the proof construction a new methodological principle may be required here. Another topic is the definition of sufficient generations for parameterised specifications. We are investigating the possibility of incorporating the idea in [18] into our mechanism.

Secondly, it is not clear whether our system can be extended to the other fragment of CafeOBJ. In the case of rewriting logic, if restricted to initial semantics, inductive proofs similar to the case of eCafeOBJ may be useful, but it is not yet certain what exact form implementations of such techniques should take. In the case of behavioural specifications, the problem is wide open.

Thirdly, the use of more powerful provers such as PVS are worth consideration. One major advantage of our system is that the proof obligations are eCafeOBJ sentences so that the user need not acquire skills in other languages and provers. Even when the system generates proof codes for the Maude-based prover, the translations are automatic, and the user need not know Maude or the prover. It is an open question whether such information hiding is possible if other provers are to be used.

Acknowledgments

We would like to thank the Maude group at SRI International, including José Meseguer, Manuel Clavel, Francisco Duran, Steven Eker, and Patrick Lincoln for many technical and social contacts on many occasions, the list of which is too long to write down here. We also appreciate technical help and guidance from Adel Bouhoula, Masayuki Fujita, Kokichi Futatsugi, Joseph Goguen, and Masami Hagiya. Finally we would like to say muchas gracias to all of the other members of the Cafe project, without whom nothing would have been realised in the current form.

REFERENCES

1. Bouhoula, A., *Automated theorem proving by test set induction* Technical Report, Computer Science Laboratory, SRI International, 1996
2. Bouhoula, A., "Using induction and rewriting to verify and complete parameterized specifications", in *Theoretical Computer Science*, Vol.170, 1996, pp.245–276
3. Bouhoula, A. and Jouannaud, J.-P., *Automata-driven automated induction*, Technical Report, Computer Science Laboratory SRI International, 1996
4. Bouhoula, A., Jouannaud, J.-P., and Meseguer, J., "Soecification and Proof in Membership Equational Logic", Technical Report, Computer Science Laboratory, SRI International, 1997
5. Clavel, M., Durán, F., Eker, S., and Meseguer, J., *Design and implementation of the Cafe prover and Church-Rosser checker tools*, Technical Report, Computer Science Laboratory, SRI International, 1998
6. Clavel, M., Durán, F., Eker, S., and Meseguer, J., "Building Equational Proving Tools by Reflection in Rewriting Logic", in this volume, Elsevier, 2000
7. Diaconescu, R. and Futatsugi, K., *Logical Semantics of CafeOBJ*, Technical Report, Japan Advanced Institute of Science and Technology, 1997
8. Diaconescu, R. and Futatsugi, K., *CafeOBJ Report - The Language, Proof Techniques, and Methodologies for Object-Oriented Algebraic Specification*, World Scientific, 1998
9. Futatsugi, K. and Nakagawa, A.T., "An overview of CAFE specification environment – an algebraic approach for creating, verifying, and maintaining formal specifications over networks –", in *The Proceedings of the 1st International Conference on Formal Engineering Methods*, IEEE, 1997, pp.170–181
10. Goguen, J. et al., *Introducing OBJ*, Technical Report SRI-SCL-92-03, Computer Science Laboratory, SRI International, 1992
11. Goguen, J., "Parameterized Programming", in *IEEE Transaction on Software Engineering* Vol.SE-10, No.5, 1984, pp.528–543

12. Kondoh, H., "Toward a Mathematical Software Engineering", to appear in *The Proceedings of FASE 2000*, Lecture Notes in Computer Science, Springer, 2000
13. Ishiguro, M., Nakagawa, A.T., Seo, A., and Futatsugi, K., "Cafe environment — formal specifications and proofs", in *The Proceedings of 20th International Conference on Software Engneering*, the Volume of Posters and Research Demonstration Sessions, IEEE, 1998
14. Meseguer, J., "Conditional Rewriting Logic as a Unified Model of Concurrency", in *Theoretical Computer Science*, Vol.93, 1992, pp.73–155
15. Meseguer, J., *Membership Algebra as a Logical Framework for Equational Specification*, Technical Report, SRI International, 1997
16. Nakagawa, A.T. et al., *CafeOBJ Manual*, Technical Report JAIST/SRA, 1997
17. Seo, A. and Nakagawa, A.T., "An Environment for Systematic Development of Algebraic Specifications on Networks", in this volume, Elsevier, 2000
18. Wirsing, M., "Algebraic Specification", in *Handbook of Theoretical Computer Science, Volume B: Formal Methods and Semantics*, Elsevier/MIT, 1990, pp.675–788

Chapter 5

Generating Rewrite Theories from UML Collaborations

Alexander Knapp*[a]

[a]Ludwig-Maximilians-Universität München
knapp@informatik.uni-muenchen.de

We present an executable model for collaborations and interactions of the "Unified Modeling Language" (UML) as a rewriting logic theory in the concurrent object-oriented algebraic specification language CafeOBJ. The static semantics of an UML collaboration, i.e. the context, is defined as a relation between the collaboration's rôles and instances modelled as CafeOBJ objects. The dynamic semantics, i.e. the semantics of interactions, is given by concurrent rewrite rules. This rewrite theory model of collaborations is directly based on UML's abstract syntax; it is proven correct with respect to a temporal logic formalisation of UML interactions.

Introduction

Simulation has proven to be a useful technique for validating interaction-centred software engineering notations, most prominently Message Sequence Charts [8], especially in the early phases of software development. The "Unified Modeling Language" (UML [2]), the emerging standard software development and description language, also comprises a notion of interaction modelling, so-called collaborations, and simultaneously features a sound abstract syntax and a rich and detailed, though informal, semantics for this concept. However, the obvious transfer of known simulation results is limited by some semantical deviations and the integration of new characteristics, since additional influences from OOSE's interactions, OMT's collaborations, and rôle modelling in general have been combined: In an UML model, collaborations specify how an operation or an use case of the model is realised by a cooperation of several instances of model elements. The static aspects of a collaboration are given by a context of class and association rôles describing which features actually participating instances have to show. For the dynamic aspects a collaboration defines an interaction, specifying which actions have to be performed by participating instances, which stimuli to other participating instances these actions have to dispatch, and in which order these stimuli can be sent sequentially or concurrently. The precise semantics of UML collaborations has been elaborated to some extent [7,12,6,1], but these investigations have not focussed on simulation of a collaboration's behaviour.

We therefore propose an executable formal model for UML collaborations that is directly based on UML's abstract syntax and provably correct with respect to the temporal logic semantics of interactions in [7], thus yielding simulation capabilities for UML's

*This work was carried out during a stay at the Computer Science Laboratory of SRI International as part of the Visitor Exchange Program P-1-3334. It was supported by a DAAD scholarship.

specific notion of interaction modelling. We use rewrite theories, more specifically the rewriting logic dimension of the algebraic specification language CafeOBJ [5], as a semantic framework that allows for a unified treatment of object-orientation and concurrency; however, other rewriting logic specification language like Maude [4] or ELAN [3] may be used just as well. Instances are modelled as CafeOBJ objects. The context of a collaboration defines a relation between instances and rôles that gives rise to proof obligations for instances that actually are taking part in a collaboration. The actions of an interaction and their execution ordering, the dynamic part of a collaboration, are interpreted as simple rewrite clauses. For the correctness proof, we interpret the generated rewrite theory as a transition system using rewriting logic. We devise a bisimilar rewrite theory containing more explicit information; fair runs inside this extended rewrite theory can be shown to be models of the linear temporal formulae schemes used in the formalisation of UML interaction's semantics.

The remainder of this paper is structured as follows: In Sect. 1 we summarise UML collaboration's concrete and abstract syntax and the formal semantics of UML interactions in temporal logic. In Sect. 2 we present a model of instances as CafeOBJ objects. Section 3 describes the generation of a CafeOBJ specification from an UML collaboration. The correctness proof for the generated rewrite theory with respect to the temporal logic specification of the collaboration's interaction is contained in Section 4. We conclude with an outlook to possible refinements and extensions.

We assume a superficial knowledge of UML's concrete and abstract syntax, see [11].

1. UML Collaborations

We briefly recall the parts of UML's concrete syntax and the UML meta-model, the abstract syntax, pertaining to collaborations and interactions [11] by means of an example; the formal semantics of interactions according to [7] is sketched in a slightly improved form in Sect. 1.3.

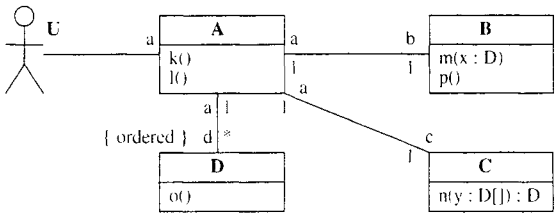
Consider the collaboration diagram in Fig. 5.1(b), which is based on the static structure diagram in Fig. 5.1(a); some relevant fragments of its abstract syntax are presented graphically in Fig. 5.2.

1.1. Contexts

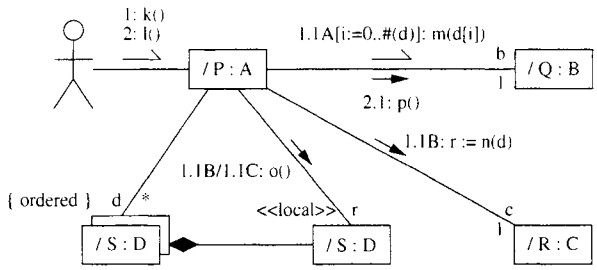
For the static aspects of a collaboration, called its context, rôles are specified that have to be filled in order to perform the task of the collaboration, describing the features collaborators have to show. Rôles are based on elements of a surrounding UML model.

In our example, the *classifier rôles*[1] `P`, `Q`, `R`, `S` (depicted as boxes), and the anonymous actor rôle (the stick figure) are based on *classes* `A`, `B`, `C`, `D` (again shown as boxes), and *actor* `U` (again a stick figure) of the static structure diagram, respectively. They have to be played by instances that show at least the features of the base *classifiers*. Analogously, the *association rôles* (straight lines) and *association end rôles* (the line ends with their annotations) defined by Fig. 5.1(b) are based on the *associations* and *association ends* (again lines and line ends, resp.) of Fig. 5.1(a) in the obvious way; they have to be played

[1] We use *italics* when introducing a category of UML's abstract syntax. As in the UML specification, the meta-classes themselves are written with initial capital letters.



(a) Static Structure Diagram



(b) Collaboration Diagram

Figure 5.1. Example of an UML Collaboration

by *links* and *link ends*. Rôle `S` is declared as multiple (stacked box) indicating that several objects can play this rôle; the aggregation (filled lozenge) to the other occurrence of `S` selects a specific instance. The association end rôle `r` is local, i.e. represents a dynamic link end inside a procedure.

1.2. Interactions

The dynamic part of a collaboration is described by messages gathered in an interaction. An interaction declares how and in which order stimuli complying to its messages are to be exchanged.

In our example, the *messages* `1`, `1.1A`, `1.1B`, `1.1C`, `2`, and `2.1` are shown (text and arrows in the middle of the straight lines). Each message refers to a sender and a receiver rôle and a communication association rôle (indicated by the direction of the arrow and the line it is next to), an *action* (extracted from the text and the arrow), predecessing messages, and an optional activating message (both extracted from the text). A message's predecessors and activator are determined by the sequence number in Dewey notation; activation is expressed by a new level and preceding by lexicographic ordering and explicitly given predecessors, concurrency of messages is indicated by capital letters; see Fig. 5.2(b). Semantically, the activator is the message that invoked the procedure which in turn in-

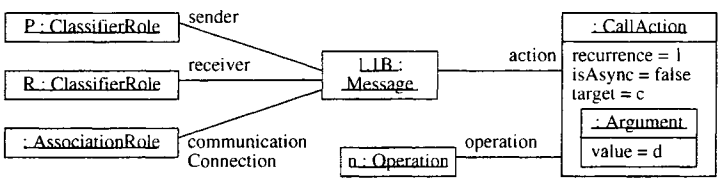
vokes the current message; before a message can be executed all its predecessor messages have to be completed. There are several constraints on these predecessor and activator relations, such that precedence forms a partial order and activation forms a forest on the set of messages M of an interaction:

i. The graph (M, P) with P the predecessor relation on M is acyclic.

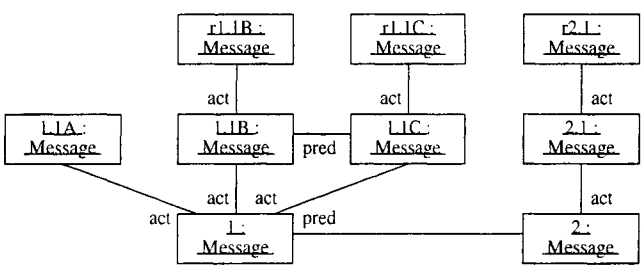
ii. The graph (M, A) with A the activator relation on M is a forest.

iii. For $m', m'' \in M$, if m'' transitively precedes m' then either both have the same activator $m \in M$ or both have no activator.

Executing a message actually means executing its action, possibly resulting in the sending of a stimulus that complies to the message and therefore its action. An Action can be a CallAction, a ReturnAction, and of several other kinds, we do not discuss here. An action has a target expression (here always implicit and to be determined by the receiver rôle of the corresponding message) that resolves to a set of target instances on evaluation; a recurrence expression (the text in square brackets or 1 if not given) that determines how the target set is iterated (sequentially or in parallel); and argument expressions that yield the actual arguments of the action. Furthermore, an action can be asynchronous (half stick arrow), i.e., on execution no results are awaited, or synchronous



(a) Abstract Syntax of Message 1.1B



(b) Abstract Syntax of the Interaction

Figure 5.2. Abstract Syntax of Fig. 5.1(b)

(filled arrow head), i.e., awaiting results. A call action refers additionally to an *operation*, a stimulus complying to such an action will call that operation with the evaluated argument expressions as actual parameters. Return actions and their messages are generally implicit (their actual arguments may be given between the colon and the assignment); moreover, a return action has no explicit target, it returns its actual arguments to a caller.

The assignment of actions to messages leads to additional consistency constraints on the set M of an interaction's messages:

iv. For $m \in M$, if the action of m is not a call then there is no $m' \in M$ such that m activates m'.

v. For $m \in M$, if, and only if the action of m is a synchronous call then there is an $r \in M$ such that the action of r is a return and m activates r and m' transitively precedes r for all $m' \in M$ such that m activates m'.

In particular, actions may be shared by several messages. Regarding an interaction as a graph (M, F) with the messages M as vertices and the activator and predecessor relations as edges F where each vertex m is labelled by $\alpha(m)$, the action of m, and each edge by whether it is an activator or a predecessor edge we require:

iv. The quotient of an interaction (M, F) by identifying messages with the same action satisfies constraints (i–v).

The intended semantics of the sample interaction can be summarised as follows: At start, only stimuli complying to `1` can occur since it is the only message that has no predecessors and no activators. Such stimuli can only be created by an execution of `1`'s action. Execution of this action means the creation of a single stimulus μ that is sent from the instance playing the actor rôle to the instance playing `P` and that bears an asynchronous call action of operation `k` with no actual arguments. Since the action of `1` is declared as asynchronous, no return stimulus is awaited and message `1` is completed; thus, stimuli complying to `2` may now occur. Furthermore, on receipt of stimulus μ, `1.1A`, `1.1B`, and `1.1C` are activated; the actions of `1.1A` and `1.1B` can now be executed concurrently, `1.1C` has to wait for the completion of `1.1B`. The action of `1.1B` calls `n` with actual argument `d`; it is synchronous, hence a return stimulus with a value for `r` is awaited and only on receipt `1.1B` is completed. On completion of `1.1B`, the action of `1.1C` can be executed analogously. However, note that `1.1B` and `1.1C` share the local link `r`; this has to be stored such that the return stimulus for `1.1B` and the stimulus for `1.1C` have access. The action of `1.1A` asynchronously calls `m` as many times as `d` has elements with actual argument `d[i]` where `i` varies with the number of calls; the message is completed when all elements of `d` are processed. Meanwhile the action of `2` may have been executed in the same vein, activating `2.1`.

1.3. Formal Semantics of UML Interactions

The dynamic semantics of interactions has been formalised in [7] using a linear temporal logic in the style of Manna and Pnueli [9]. A state variable yields the global system state on which an interaction is performed; functions for local states of actions and stimuli sent and received partially characterise this state. Execution of actions is described by

transition systems on the local states of actions. Temporal formulae over the system state constrain the concurrent execution of actions. We briefly review this semantics with respect to which we will prove the generated rewrite theories as correct.

The following semantic domains are required: A domain Σ of global system states representing the states of instances playing the different rôles in the context of an interaction; for each action a a domain Λ_a of local action states comprising information from the recurrence and the target expression of a; for each action a a domain M_a for stimuli complying to a; and for each message m a domain M_m for stimuli complying to m and the action $\alpha(m)$ attached to it, such that there is a map $\alpha : \mathrm{M}_m \to M_{\alpha(m)}$ exhibiting information that is shared by several stimuli. The semantic domain Σ is equipped with maps

$$a_p : \Sigma \to \{\bot, \downarrow\} \uplus (\mathcal{N} \times \wp\mathcal{N} \times *_{\dashv}) \qquad \text{and}$$
$$m_p : \Sigma \to \{\bot, \downarrow\} \uplus \mathrm{M}_m$$

for every action a, every message m, and every $p \in (\mathcal{N} \times \mathcal{N})^+$. The number pair sequences p are used to distinguish different occurrences of actions and stimuli; nesting is reflected by the length of a sequence, the pairs reflect the possibly parallel occurrences on a given nesting level. Intuitively, the map a_p either yields the local action state of the pth occurrence of a together with a "program counter" that is used to create stimuli and a set of message numbers whose return is awaited; or a_p yields that the pth occurrence of a is undefined ($\bot$) in a system state; or has already terminated ($\downarrow$). Analogously, the pth stimulus occurrence for a message m has been sent but not yet received if m_p yields an element of M_m; this stimulus has not been sent if m_p yields $\bot$; and this stimulus has already been received if m_p yields $\downarrow$.

The semantics of each action a is given by some initial local action state depending on the system state and a sequence number $p \in (\mathcal{N} \times \mathcal{N})^+$

$$\lambda_a^p : \Sigma \to \Lambda_a$$

and transition relations

$$\rightarrow_a^\sigma \subseteq (\Lambda_a \times (\{\downarrow\} \uplus \Lambda_a)) \uplus (\Lambda_a \times (\Lambda_a \times \wp\mathrm{M}_a))$$

parameterised over the global system state $\sigma \in \Sigma$.

Action occurrences will be created in an initial state. After creation, an action occurrence may proceed either by terminating and we write $\lambda\downarrow_a^\sigma$ if $(\lambda, \downarrow) \in \rightarrow_a^\sigma$; or it may proceed by a silent step changing only its local state and we write $\lambda \rightarrow_a^\sigma \lambda'$ if $(\lambda, \lambda') \in \rightarrow_a^\sigma$ with $\lambda' \neq \downarrow$; or it may proceed by changing its local state and sending stimuli and we write $\lambda \rightarrow_a^\sigma \lambda', \mathrm{M}$ if $(\lambda, (\lambda', \mathrm{M})) \in \rightarrow_a^\sigma$. An action's recurrence expression may allow for sending several different stimuli in a single transition step.

The semantics of full interactions and their ordering constraints uses a linear first-order temporal logic with temporal connectives $\Box$ (always), $\Diamond$ (eventually), and W (unless). The underlying state language consists of one flexible state variable σ from the semantic domain Σ, rigid variables, the semantic maps a_p and m_p as function symbols, and $\rightarrow_a^\sigma$ as relation symbols (we also use the various abbreviations introduced above): Let I be

$$\forall p \in (\mathcal{N} \times \mathcal{N})^+ . \dashv_{\surd}(\sigma) = \bot \text{ W } \dashv_{\surd}(\sigma) = (\prime, \emptyset, \lambda^{\surd}_{\dashv}(\sigma)) \tag{5.1}$$

$$\begin{aligned}
&\Box(a_p(\sigma) = (i, W, \lambda) \supset (a_p(\sigma) = (i, W, \lambda) \text{ W} \\
&\quad ((a_p(\sigma) = (i, W \setminus \{j\}, \lambda) \wedge j \in W) \vee \\
&\quad (a_p(\sigma) = \downarrow \wedge \lambda \downarrow^{\sigma}_{a} \wedge W = \emptyset) \vee \\
&\quad (a_p(\sigma) = (i, W, \lambda') \wedge (\lambda \rightarrow^{\sigma}_{a} \lambda') \wedge W = \emptyset) \vee \\
&\quad (a_p(\sigma) = (i+1, wait_a(k), \lambda') \wedge (\lambda \rightarrow^{\sigma}_{a} \lambda' . \{\mu_0, \ldots, \mu_k\}) \wedge W = \emptyset \wedge \\
&\quad \bigwedge_{\substack{0 \le j \le k \\ m \in \alpha^{-1}(a)}} \alpha(m_{p.(i,j)}(\sigma)) = \mu_j \wedge \forall j > k . m_{p.(i,j)}(\sigma) = \bot)))
\end{aligned} \tag{5.2}$$

$$\Box(a_p(\sigma) = \downarrow \supset \Box a_p(\sigma) = \downarrow) \tag{5.3}$$

$$m_{p.(i,j)}(\sigma) = \bot \text{ W } (\alpha(m)_p(\sigma) = (i+1, W, \lambda) \wedge m_{p.(i,j)}(\sigma) \neq \bot) \tag{5.4}$$

$$\Box((m_p(\sigma) = \mu \wedge \mu \neq \bot) \supset (m_p(\sigma) = \mu \text{ W } m_p(\sigma) = \downarrow)) \tag{5.5}$$

$$\Box(m_p(\sigma) \neq \bot \supset \Diamond m_p(\sigma) = \downarrow) \tag{5.6}$$

$$\Box(\alpha(m')_p(\sigma) = (0, \emptyset, \lambda) \supset \alpha(m)_p(\sigma) = \downarrow), \qquad \text{if } m \longrightarrow m' \tag{5.7}$$

$$\Box(\alpha(m')_{p.(i,j)} = (0, \emptyset, \lambda) \supset m_{p.(i,j)}(\sigma) = \downarrow), \qquad \text{if } m \dashrightarrow m' \tag{5.8}$$

$$\Box(m_{p.(i,j)}(\sigma) = \downarrow \supset \Diamond \alpha(m')_{p.(i,j)}(\sigma) = (0, \emptyset, \lambda)), \qquad \text{if } m \dashrightarrow m' \tag{5.9}$$

$$\Box(m'_{p.(i,j).(k,l)}(\sigma) = \mu \wedge \mu \neq \bot, \downarrow \supset \alpha(m)_p(\sigma) = (i+1, W, \lambda) \wedge j \in W), \quad \text{if } m \dashrightarrow m' \text{ and } m' \text{ return} \tag{5.10}$$

$$\forall m \in M . \forall p \in (\mathcal{N} \times \mathcal{N})^+ \setminus \{(\prime, \prime)\} . \alpha(\Updownarrow)_{\surd}(\sigma) = \bot \tag{5.11}$$

$$\forall m \in M \setminus N . \alpha(m)_{(0.0)}(\sigma) = \bot \tag{5.12}$$

$$\forall m \in N . \Diamond \alpha(m)_{(0.0)}(\sigma) = (0, \emptyset, \lambda^{(0,0)}_{\alpha(m)}(\sigma)) \tag{5.13}$$

Table 5.1
Formal Semantics of Interactions

an interaction, M its set of messages, A the set of actions that is attached to M, and let $\alpha(m)$ denote the action of $m \in M$. We write $m \dashrightarrow m'$ if message m is the activator message of m' and $m \longrightarrow m'$ if message m is a predecessor message of m'; let $N \subseteq M$ be the set of messages without an activator message; the functions $wait_a : \mathcal{N} \rightarrow \wp\mathcal{N}$ are defined as $wait_a(n) = \{0, \ldots, n\}$ if a is a synchronous call action and $wait_a(n) = \emptyset$ otherwise. The *semantics* of I is defined to be all runs (models) of the temporal logic specification yielded by instantiating formula schemes (5.1–5.13) in Table 5.1 according to I.

2. Semantic Domains for Instances

Instances playing a rôle in a collaboration belong to classifiers. Therefore, we first define a translation of classifiers, associations, and association ends to CafeOBJ, providing the semantic domains for instances. (We restrict ourselves to binary associations and simple association ends without qualifiers; furthermore we omit call-by-reference parameters of operations of the kinds `pdk-out` and `pdk-inout`.)

The translation is illustrated by examples from the static structure diagram in Fig. 5.1(a). Here, we refer to the complete abstract syntax of this static structure diagram in App. B given as a configuration of objects according to the UML meta-model in App. A. This abstract syntax introduces unique names for all model elements such that the association between classes `A` and `B` bears name `AB`, the operation of class `C` bears name `C-n`, and so forth.

We model classifiers directly as CafeOBJ classes; in particular, operations are treated separately. Associations and association ends are not given a first-class status, but opposite association ends of a classifier are modelled like attributes. For every class or actor C we define

```
class C < D1 ... Dp { at1 : τ1 ... atm : τm ae1 : τ'1 ... aen : τ'n }
[ CObjectId < ObjectId ]
[ CObjectId < D1ObjectId ... DpObjectId ]
```

where each at_i is the name of an attribute of C, each ae_j the name of a navigable association end opposite of C, and each D_k is a direct super-classifier of C. The CafeOBJ sorts τ_i and τ'_j are determined as follows: For an attribute at_i with type ty_i we set $\tau_i = C'$`ObjectId` where ty_i is the name of a classifier C'; if ty_i is the name of a datatype we set $\tau_i = ty_i$ (assuming that such a datatype has been suitably axiomatised in CafeOBJ); if ty_i is of the form $name$`[]` we set $\tau_i =$ `List[`τ`]` and τ is determined from $name$ as before. For an opposite association end ae_j with type ty_j we proceed as for attributes if the multiplicity of ae_j is `1`; if this multiplicity is `*`, let τ_j be determined as if the multiplicity would be `1` and set $\tau'_j =$ `List[`τ_j`]` if the association end is ordered and $\tau'_j =$ `Set[`τ_j`]` if the association end is unordered.

For example, classifier `A` becomes

```
class A { UA-u : AObjectId  AB-b : BObjectId  AC-c : CObjectId
          AD-d : List[DObjectId] }
[ AObjectId < ObjectId ]
```

Operations are modelled by messages of CafeOBJ. For each operation O with input parameters, i.e. of kind `pdk-in`, with types $ty_1, \ldots, ty_m$ (in this order) and return parameters, i.e. of kind `pdk-return` of types $ty'_1, \ldots, ty'_n$ (again in this order) we define

```
op O : ObjectId τ1 ... τm ObjectId -> Message
op return-O : ObjectId τ'1 ... τ'n -> Message
```

where the CafeOBJ sorts τ_i and τ'_j are determined as for attributes. The first parameter will hold the object identifier of the sender instance, the last the object identifier of the receiver. We uniformly provide return messages for every operation; they only bear the object identifier of the sending instance.

For example, operation `C-n` becomes

```
op C-n : ObjectId List[DObjectId] ObjectId -> Message
op return-C-n : ObjectId DObjectId -> Message
```

3. Generating Rewrite Theories from Collaborations

We generate a CafeOBJ specification from a given collaboration. The collaboration's context which defines the rôles that have to be played in the collaboration generates a

cast relation between actual instances and rôle names. This relation gives rise to proof obligations that, if an instance plays a rôle, the actual classifier of the instance shows all features required by the rôle. Messages and actions of the collaboration's interaction generate rewrite rules. The ordering of the execution of actions and the sending of stimuli is also expressed by these rewrite rules. For simplicity, we restrict ourselves to the case where no actions are shared between messages and where messages are not sent concurrently.

Again, we accompany the description of the generation process by some examples from the collaboration in Fig. 5.1(b) and its abstract syntax in App. B.

3.1. Contexts

An instance playing a classifier rôle in a collaboration has to show all the rôle's features, structural as well as behavioural, but does not have to belong to a certain class; analogously, links and link ends only have to show the properties of the association and association end rôles that they are playing. However, an instance may play different rôles in a collaboration. For the execution of a collaboration a cast relation, which instance is playing which classifier rôle, etc., has to be fixed. The corresponding proof obligations can be discharged on the basis of the UML model or on the basis of the CafeOBJ translation for the model's static structure. In particular, it has to be shown that a cast element that is used as an argument of an operation call has the required type.

The cast relation has the following signature:

```
[ CastElement ]
op _plays_ : AttrId Name -> CastElement

[ CastSet ]
[ CastSet < CastList ]
op empty : -> CastSet
op _,_ : CastSet CastSet -> CastSet { assoc comm id: empty }

[ Cast ]
[ Cast < ACZ-Configuration ]
op _plays_|_ : ObjectId Name CastSet -> Cast
```

For a collaboration, a configuration of CafeOBJ objects and messages together with a cast relation complying to the collaboration forms the semantics of the context. A cast relation complies to a collaboration's context, if all rôles are played by sufficiently many instances and link ends according to the multiplicity of the rôles.

For the sample collaboration in Fig. 5.1(b), such a configuration could be

```
< ou : U | UA-a = oa > (ou plays U-Role | UA-a plays UP-a)
< oa : A | UA-u = null, AB-b = ob, AC-c = oc,
           AD-d = (od1 :: od2 :: od3 :: od4 :: nil) >
  (op plays P | UA-u plays UP-u, AB-b plays PQ-b,
                AC-c plays PR-c, AD-d plays PS-d)
< ob : B | AB-a = null > (ob plays Q | AB-a plays PQ-a)
< oc : C | AC-a = null > (oc plays R | AC-a plays PR-a)
< od1 : D | AD-a = null > (od1 plays S | AD-a plays PS-a)
< od2 : D | AD-a = null > (od2 plays S | AD-a plays PS-a)
```

```
< od3 : D | AD-a = null > (od3 plays S | AD-a plays PS-a)
< od4 : D | AD-a = null > (od4 plays S | AD-a plays PS-a)
```

If only instances of a single class and link ends of a single association end, on which the corresponding rôles are based, are to play these rôles, the cast relation may be omitted. In the following, to ease the presentation, we assume that this is the case for all association end rôles.

3.2. Interactions

The communication in collaborations is triggered by actions which have to be executed according to the activation and predecessor relation of their messages. We first introduce some semantic domains for actions. The inter-operation of these structures is laid down in computational rules.

Semantic Domains for Actions

We define a CafeOBJ class as semantic domain for actions.

```
class Action { ctxt : ActivationObjectId  wait : Bool
               target : Set[ObjectId] }
[ ActionObjectId < ObjectId ]
```

Objects of class `Action` have targets derived from the target attribute of the meta-class Action, a wait flag, that is used for synchronous actions awaiting a return, and an *activation context.* The notion of activation context is motivated as follows: By [11, p. 2-102], every action has to be executed within the context of an instance. Additionally, [11, p. 3-104] suggests that every action adherent to a message that is activated by another message is to be executed as part of an activation, itself adhering to an instance. and that such an activation holds local information. We thus introduce a semantic domain for activation contexts, viz. objects of

```
class Activation { obj : ObjectId  ret : ObjectId  succ : Set[ClassId] }
[ ActivationObjectId < ObjectId ]
```

where `obj` refers to the object modelling the instance that the activation belongs to, `ret` refers to the action which created the current activation, and `succ` holds the class names of the actions that have to be activated inside the activation; thus, activation contexts reflect the block structure of calls.

For each message of the interaction that has reference to a call action we define a subclass of `Activation` that additionally holds the association ends that are local to this message, i.e., association ends stereotyped ≪local≫, and attributes for the actual arguments of the activating message's action. We also require an initial activation context, which forms the context of all actions whose messages have no activator and which has to hold local association ends that are not assigned to any particular message; if the collaboration models the realisation of an use case, this activation context does not belong to an instance.

For example, the association end rôle `r` is local to message `m1` and therefore we define

```
class Activation-m1 < Activation { r : ObjectId }
[ Activation-m1-ObjectId < ActivationObjectId ]
```

For message `m1-1B` we define

```
class Activation-m1-1B < Activation { C-n-y : List[DObjectId] }
[ Activation-m1-1B-ObjectId < ActivationObjectId ]
```

The initial activation context is

```
class Activation-init < Activation { }
[ Activation-init-ObjectId < ActivationObjectId ]
```

The semantic domain `Action` is also further specialised: For each message m of the given interaction we devise a subclass of `Action`, representing the local state space of the message's action a:

```
class Action-m < Action { rc1 : τ1 ... rcn : τn }
[ Action-m-ObjectId < ActionObjectId ]
```

where the attributes rc_i and sorts τ_i have to be derived from the recurrence attribute of a. More precisely, if the recurrence expression of a is `1`, we set $n = 0$; if it is of the form `iterate[`*name*`, ..., ...]`, i.e. the abstract representation of, say, `[i:=0..#(d)]`, we set $n = 1$ and $rc_1 =$ *name*, and $\tau_1 =$ `Nat`. Numerous other iteration expressions may be conceived.

For example, the action adherent to message `1.1A` becomes

```
class Action-m1-1A < Action { i : Nat }
[ Action-m1-1A-ObjectId < ActionObjectId ]
```

Action and activation objects will be created dynamically during the execution of a collaboration. This will be achieved by a generic new object identifier function

```
op new : ClassId Nat -> ObjectId
```

This function cooperates with class objects that represent classes on the object level, like in [10], and count how many instances of a certain CafeOBJ class have been created already:

```
class ProtoObject { class : ClassId  cnt : N }
[ ProtoObjectObjectId < ObjectId ]
```

For each CafeOBJ class we require such a class object to exist in an actual configuration. Furthermore, we are using a null identifier for objects

```
op null : -> ObjectId
```

Finally, stimuli are not directly represented by CafeOBJ messages for operations, but by wrapped messages in the following way:

```
op call-m : ActionObjectId Message -> Message
op return : Message ActionObjectId -> Message
```

A call stimulus for a message m additionally carries information which action originated this stimulus; a return stimulus is not sent to a target instance but rather back to an action.

The *system state* is given by objects that play rôles of the collaboration together with the cast relation, undelivered stimuli carrying operation calls and returns, action objects, and some activations; it is summarised in a `ACZ-Configuration`.

Computational Rules

The dynamic behaviour of actions is modelled by rewrite rules on the system state. These rules take account of the type of actions, the iteration of actions, whether actions are synchronous or asynchronous, and of the execution ordering. We make use of the following notational conventions: $\overrightarrow{X}$ designates $X_0, \ldots, X_n$ where the iterator is clear from the context; $[X]$ means that X has to be omitted under some circumstances which are stated with the rules.

Given an object `O` playing the sender rôle S and an activation context `G` we may start any action that is to be done as part of this activation `G`, provided that all predecessor messages have already been finished. More precisely, when `G` is an activation context for the action corresponding to message m' (or the initial one), an action corresponding to message m with predecessors $d_1, \ldots, d_n$ that is activated by m' can be started if $d_1, \ldots, d_n$ have been finished in this context; a message has been finished if the target attribute of its action is empty and if this action is not waiting for a return. Furthermore, the new action must not have been started before, that is, its class name must occur in the `succ` attribute of the current activation. The target set and the recurrence attributes of a new action have to be suitably initialised. The necessary information is provided by the action's target expression t and its recurrence expression r; the first has to be evaluated completely, $[\![t]\!]$, whereas from the latter only initialisations have to be extracted, $[\![r]\!]_{\mathrm{i}}$. Hence,

```
crl [start-m]:
  < D : Action-d | ctxt = G, wait = false, target = empty >
  < O : C | b = V > (O plays S | )
  < G : Activation-m' | [obj = O,] l = L, succ = S >
  < H : ProtoObject | class = Action-m, cnt = N > =>
    < D : Action-d >
    < O : C | b = V > (O plays S | )
    < G : Activation-m' | succ = S' >
    < H : ProtoObject | class = Action-m, cnt = N+1 >
    < new(Action-m, N) : Action-m |
        ctxt = G, wait = false, target = [[t]], rc = [[r]]i >
  if < Action-m, S' > = choose(S) .
```

for every message m. The function `choose` splits a set into one of its elements and the remaining set. If m' is `init` and the collaboration realises an use case, `obj = O`, has to be left out since then the initial context does not belong to a particular instance.

Call actions are performed step by step according to the corresponding recurrence expression; we assume that the semantics of a recurrence expression r, on the one hand, is given by a function $[\![r]\!]$ that yields a pair consisting of an object identifier and a new target set and that the returned object identifier is `null` if no call is to be done in this step (otherwise adding this object identifier to the new target set the original target set is regained); on the other hand, we require functions $[\![r]\!]_i$ updating the recurrence attributes according to r. If the action is synchronous, a return is awaited and thus the wait flag is set; otherwise the wait flag remains unchanged. Hence,

```
crl [do-m]:
```

```
< O : C | b = V > (O plays S | )
< G : Activation-m' | [obj = O,] l = L >
< A : Action-m | ctxt = G, wait = false, target = Ts > =>
  < O : C > (O plays S | )
  < G : Activation-m' >
  if T =/= null
  then < A : Action-m | target = Ts', [wait = true,] rc = [[r]] >
       call-m(A, O(O, [[e]], T))
  else < A : Action-m | target = Ts' > fi
if Ts =/= empty and < T, Ts' > = [[r]] .
```

for every message m with a call action, where e_i are the actual arguments of m's action and O its operation.

In contrast, return actions are only performed once in a given activation; they do not have an explicit target expression, but use the return attribute of the activation in which they are executed. Hence,

```
crl [do-m]:
  < O : C | b = V > (O plays S | )
  < G : Activation-m' | [obj = O], l = L, ret = R >
  < A : Action-m | ctxt = G > =>
    < O : C > (O plays S | )
    < G : Activation-m' | ret = null >
    < A : Action-m >
    return(return-O(O, [[e]]), R)
  if R =/= null .
```

for every message m with a return action, where e_i are the actual arguments of m's action and O is the operation of m's activator action.

On receipt of a call stimulus at an object, a new activation context for this object has to be created; actions whose messages $m_0, \ldots, m_n$ are to be activated by the current stimulus, that is inside the new activation context, may then be started. The local attributes of the new activation context are filled with information provided by the parameters of the call stimulus, $[\![\mathrm{P}]\!]_i$, according to the specification of the collaboration. Hence,

```
rl [receive-m]:
  call-m(A, O(S, P, O))
  < O : C | b = V > (O plays S | )
  < H : ProtoObject | class = Activation-m, cnt = N > =>
    < O : C > (O plays S | )
    < H : ProtoObject | class = Activation-m, cnt = N+1 >
    < new(Activation-m, N) : Activation-m |
       obj = O, l = [[P]], ret = A, succ = { Action-m } > .
```

for every message m with a call action.

A return stimulus of message m reactivates the execution of message m''s action, such that m activates m'; the action of m' itself runs in an activation context generated by message m'', or the initial activation context, if m' has no activator. This activation context has to be updated according to the parameters of the return stimulus, $[\![\mathrm{P}]\!]_i$. Hence,

```
rl [receive-m]:
  return(return-O(S, P->), A)
  < O : C | b = V-> > (O plays S | )
  < G : Activation-m'' | [obj = O,] l = L-> >
  < A : Action-m' | ctxt = G, wait = true > =>
    < O : C > (O plays S | )
    < G : Activation-m'' | l = [[P]]-> >
    < A : Action-m' | wait = false > .
```

for every message m with a return action.

We illustrate this generation procedure by the rules for message `m1-1B` and its corresponding return message `r1-1B`.

```
crl [start-m1-1B]:
  < O : C | AC-c = C' > (O plays P | )
  < H : ProtoObject | class = Action-m1-1B, cnt = N >
  < G : Activation-1 | obj = O, succ = S > =>
    < O : C > (O plays P | )
    < H : ProtoObject | class = Action-m1-1B, cnt = N+1 >
    < G : Activation-1 | obj = O, succ = S >
    < new(Action-m1-1B, N) : Action-m1-1B |
        ctxt = G, wait = false, target = makeSet(1, C') >
  if < Action-m1-1B, S' > = choose(S) .
crl [do-m1-1B]:
  < O : C | AD-d = D1 > (O plays P | )
  < G : Activation-1 | obj = O >
  < A : Action-m1-1B | ctxt = G, wait = false, target = Ts > =>
    < O : C > (O plays P | )
    < G : Activation-1 >
    if T =/= null then
    then < A : Action-m1-1B | target = Ts', wait = true >
         call-m1-1B(A, C-n(O, D1, T))
    else < A : Action-m1-1B | target = Ts' > fi
  if Ts =/= empty and < T, Ts' > = choose(Ts) .
rl [receive-m1-1B]:
  call-m1-1B(A, C-n(S, P1, O))
  < O : C > (O plays R | )
  < H : ProtoObject | class = Activation-m1-1B, cnt = N > =>
    < O : C > (O plays R | )
    < H : ProtoObject | class = Activation-m1-1B, cnt = N+1 >
    < new(Activation-m1-1B, N) : Activation-m1-1B |
        obj = O, C-n-y = P1, ret = A, succ = { Action-r1-1B } > .
crl [do-r1-1B]:
  < O : C > (O plays R | )
  < G : Activation-m1-1B | obj = O, C-n-y = Y1, ret = R >
  < A : Action-r1-1B | ctxt = G > =>
    < O : C > (O plays R | )
    < G : Activation-m1-1B | ret = null >
```

```
    < A : Action-r1-1B >
    return(return-C-n(O, hd(Yl)), R)
  if R =/= null .
rl [receive-r1-1B]:
  return(return-C-n(S, Q), A)
  < O : C > (O plays P | )
  < G : Activation-1 | obj = O, r = R >
  < A : Action-m1-1B | ctxt = G, wait = true > =>
    < O : C > (O plays P | )
    < G : Activation-1 | r = Q > .
    < A : Action-m1-1B | wait = false > .
```

Start Configurations

Apart from a cast relation, fixing which instances and which links play which rôles, a collaboration's start configuration has to show an initial activation in which the collaboration's interaction can be executed.

In our example, a start configuration may thus look as follows:

```
< ou : U | UA-a = oa > (ou plays U-Role | ) ...
< od4 : D | AD-a = null > (od4 plays S | )
< g0 : Activation-init | obj = null, ret = null,
                         succ = { Action-m1, Action-m2 } >
```

4. Correctness of the Translation

We prove the translation of collaborations to CafeOBJ described above to be correct with respect to the formal semantics of interactions in Sect. 1.3. The CafeOBJ specification generated from a collaboration defines a rewrite theory and thereby, in terms of rewriting logic, a transition system with configurations as states and rewrites as transitions. This transition system indeed is a model of the temporal formulae that describe the formal semantics of this collaboration when employing a suitable interpretation of the function and relation symbols used.

Given a rewrite theory $\Xi = ((\Omega, \Gamma), \mathrm{R})$ with (Ω, Γ) being an order-sorted equational specification and R a set of rewrite rules of the form $l : t \longrightarrow t' \Leftarrow \Pi$, deduction, i.e. rewriting, takes place according to rewriting logic defined by the following four rules, cf. [10]:

1. Reflexivity.

$$\frac{}{t \xrightarrow{t} t}$$

2. Congruence. For each function symbol $f : s_1 \ldots s_n \longrightarrow s \in \Omega$

$$\frac{t_1 \xrightarrow{\alpha_1} u_1, \ \ldots, \ t_n \xrightarrow{\alpha_n} u_n}{f(t_1, \ldots, t_n) \xrightarrow{f(\alpha_1, \ldots, \alpha_n)} f(u_1, \ldots, u_n)}$$

3. Replacement. For each rewrite rule $l : t \longrightarrow t' \Leftarrow \Pi$ in R

$$\frac{t_1 \xrightarrow{\alpha_1} u_1, \ \ldots, \ t_n \xrightarrow{\alpha_n} u_n}{t_0(t_1, \ldots, t_n) \xrightarrow{l(\alpha_1, \ldots, \alpha_n)} u_0(u_1, \ldots, u_n)}, \quad \text{if } \Pi(t_1, \ldots, t_n)$$

4. Composition.

$$\frac{t_1 \xrightarrow{\alpha_1} t_2,\ t_2 \xrightarrow{\alpha_2} t_3}{t_1 \xrightarrow{\alpha_1;\alpha_2} t_3}$$

where matching is defined modulo the equations Γ. The proof terms, i.e. the deduction relation's inscriptions, reflect the structure of the rewrites and the term structure.

As is well known [10], any deduction $t \xrightarrow{\alpha} t'$ in Ξ can be transformed to a sequence of one-step rewrites $t = t_0 \xrightarrow{\alpha_0} t_1 \xrightarrow{\alpha_1} \ldots \xrightarrow{\alpha_{n-1}} t_n = t'$ such that every $\alpha_0, \ldots, \alpha_{n-1}$ contains exactly one rewrite rule label. Disregarding the term structure we only consider rewrite rule labels as proof terms; we additionally use a special label id to express reflexivity. These one-step rewrites of Ξ obviously define a transition system. A *one-step run* of Ξ is a possibly infinite rewrite sequence $t_0 \xrightarrow{l_0} t_1 \xrightarrow{l_1} t_2 \xrightarrow{l_2} \ldots$ where each deduction step is a one-step rewrite. A one-step run $t_0 \xrightarrow{l_0} t_1 \xrightarrow{l_1} t_2 \xrightarrow{l_2} \ldots$ is *fair*, if an infinite tail $t_k \xrightarrow{\mathrm{id}} t_{k+1} \xrightarrow{\mathrm{id}} t_{k+2} \xrightarrow{\mathrm{id}} \ldots$ implies that no rewrite rule in R can be applied to t_k.

In order to verify the CafeOBJ specification Ξ generated from a given collaboration C with respect to its formal semantics, we consider a slightly more complicated rewrite system for C that, in fact, is bisimilar to Ξ. Each `Action` and `Activation` class and also each message is provided with a unique identifier, a sequence of numbers that reflects the activation ordering like the temporal logic formalisation's function symbols; additionally, messages are not destroyed when being received, but rather bear a boolean flag that is switched to true on receipt.

```
class Action { ctxt : ActivationObjectId  wait : Bool
               target : Set[ObjectId]  cnt : Seq[Nat] }

class Activation { obj : ObjectId  ret : ActionObjectId
                   succ : Set[ClassId]  cnt : Seq[Nat] }

op call-m : ActionObjectId Message Seq[Nat] Bool -> Message
op return : Message ActionObjectId Seq[Nat] Bool -> Message
```

The generation process of rewrite rules is changed accordingly, in particular:

```
crl [start-m]:
  < D : Action-d | ctxt = G, target = empty, wait = false >
  < O : C | b = V > (O plays S | )
  < G : Activation-m' | [obj = O,] cnt = Q, l = L, succ : S >
  < H : ProtoObject | class = Action-m, cnt = N > =>
    < D : Action-d >
    < O : C | b = V > (O plays S | )
    < G : Activation-m' | succ : S' >
    < H : ProtoObject | class = Action-m, cnt = N+1 >
    < new(Action-m, N) : Action-m |
        ctxt = G, wait = false, target = [[e]], cnt = Q.0, rc = [[r]]_i >
  if < Action-m, S' > = choose(S) .

crl [do-m]:
  < O : C | b = V > (O plays S | )
  < G : Activation-m' | [obj = O,] l = L >
```

```
< A : Action-m | ctxt = G, wait = false, target = Ts, cnt = Q > =>
  < O : C > (O plays S | )
  < G : Activation-m' >
  if T =/= null
  then < A : Action-m | target = Ts', [wait = true,] rc = [[r]],
                        cnt = Q+1 >
       call-m(A, O(O, [[e]], T), Q, false)
  else < A : Action-m | target = Ts' > fi
if Ts =/= empty and < T, Ts' > = [[r]] .
```

where we assume `Seq[Nat]` to specify that $Q.N$ extends number sequence Q by a new level with entry N and that Q+1 adds one to the last entry of sequence Q.

For a given collaboration C, the rewrite theories Ξ, generated by the procedure described in Sect. 3, and the rewrite theory Ξ', generated by the extended procedure above, are bisimilar; for the relation $\sim$ between configurations of Ξ and configurations of Ξ' such that $E \sim E'$ if $\pi(E') = E$ where π erases all `cnt` attributes from subclasses of `Action` and `Activation`, erases messages of the form `call-`m`(`A`, `M`, `Q`, true)` and `return(`M`, `A`, `Q`, true)`, and also replaces `call-`m`(`A`, `M`, `Q`, false)` by `call-`m`(`A`, `M`)` and `return(`M`, `A`, `Q`, false)` by `return(`M`, `A`)` is a bisimulation between the transition systems defined by Ξ and Ξ'.

Thus, let C be a collaboration with interaction I and $\Xi = ((\Omega, \Gamma), \mathrm{R})$ be the rewrite theory generated from C and I by the extended procedure above.

We choose the semantic domain Σ as the set of all ground terms of the sort `ACZ-Configuration` such that all objects of the same `Action` and `Activation` class have different `cnt` attributes, respectively, all `call-`m messages having object identifiers of the same `Action` class as their first argument have different third arguments, and, finally, all `return` messages having object identifiers of the same `Action` class as their second argument have different third arguments; thus all entities have *unique number sequences*.

For each action a and its corresponding message m in I we define

$$\Lambda_a = \{\texttt{<}\ A\ \texttt{:}\ \texttt{Action-}m\ \texttt{|}\ \overrightarrow{b\ \texttt{=}\ V}\ \texttt{>} \mid \texttt{<}\ A\ \texttt{:}\ \texttt{Action-}m\ \texttt{|}\ \texttt{wait}\ \texttt{:}\ B,\ \overrightarrow{b\ \texttt{=}\ V}\ \texttt{>} \in \Sigma\}\ .$$

and

$$\mathrm{M}_a = \{\texttt{call-}m\texttt{(}A\texttt{,}\ M\texttt{,}\ Q\texttt{,}\ F\texttt{)} \mid \texttt{call-}m\texttt{(}A\texttt{,}\ M\texttt{,}\ Q\texttt{,}\ F\texttt{)} \in \Sigma,\ A : \texttt{Action-}m\texttt{-ObjectId}\}\ ,$$

if a is a call action, and

$$\mathrm{M}_m = \{\texttt{return(}M\texttt{,}\ A\texttt{,}\ Q\texttt{,}\ F\texttt{)} \mid \texttt{return(}M\texttt{,}\ A\texttt{,}\ Q\texttt{,}\ F\texttt{)} \in \Sigma,\ A : \texttt{Action-}m'\texttt{-ObjectId}\}\ ,$$

if a is a return action and the activator of m is message m'. Furthermore, for each message m in I we set

$$\mathrm{M}_m = \mathrm{M}_a\ , \qquad \alpha(\mu) = \mu\ .$$

We equip the domain of system states with maps $a_p : \Sigma \to \{\bot, \downarrow\} \uplus (\mathcal{N} \times \wp\mathcal{N} \times *_{\neg})$ and $m_p : \Sigma \to \{\bot, \downarrow\} \uplus \mathrm{M}_m$ for every action a, message m, and sequence number $p \in (\mathcal{N} \times \mathcal{N})^+$:

Given a $\sigma \in \Sigma$ and a $q \in \mathcal{N}^+$ we say that the qth occurrence of message m's action a exists in σ, if some $< A : \texttt{Action-}m \mid \texttt{cnt} = q, \ldots > \in \Lambda_a$ is part of σ; analogously, we say that the qth occurrence of a stimulus complying to a message m exists in σ, if some $\texttt{call-}m(A, M, q, F) \in \mathrm{M}_m$ or $\texttt{return}(M, A, q, F) \in \mathrm{M}_m$ is part of σ. With these abbreviations,

$$a_p(\sigma) = \bot \ ,$$

if $p = (k_0, l_0) \ldots (k_n, l_n)$ with $l_i \geq 1$ for an $0 \leq i \leq n$, or $p = (k_0, 0) \ldots (k_n, 0)$ but the $k_0 \ldots k_n$th occurrence of a does not exist in σ;

$$a_p(\sigma) = \downarrow \ ,$$

if $p = (k_0, 0) \ldots (k_n, 0)$ and the $k_0 \ldots k_n$th occurrence of a is of the form $< A : \texttt{Action-}m \mid \texttt{target = empty, wait = false, } \ldots >$;

$$a_p(\sigma) = (N, W, < A : \texttt{Action-}m \mid \overrightarrow{b = V} >) \ ,$$

if $p = (k_0, 0) \ldots (k_n, 0)$ and $< A : \texttt{Action-}m \mid \overrightarrow{b : V} >$ is the $k_0 \ldots k_n$th occurrence of a in σ with its cnt attribute set to $Q.N$, but its target attribute not set to empty or its wait attribute not set to false, and $W = \{0\}$ if wait is true and $W = \emptyset$ otherwise. Analogously,

$$m_p(\sigma) = \bot \ ,$$

if $p = (k_0, l_0) \ldots (k_n, l_n)$ with $l_i \geq 1$ for an $0 \leq i \leq n$, or $p = (k_0, 0) \ldots (k_n, 0)$ but the $k_0 \ldots k_n$th occurrence of a stimulus complying to m does not exist in σ;

$$m_p(\sigma) = \downarrow \ ,$$

if $p = (k_0, 0) \ldots (k_n, 0)$ and the $k_0 \ldots k_n$th occurrence of a stimulus complying to m is of the form $\texttt{call-}m(A, M, Q, \texttt{true})$ or $\texttt{return}(M, A, Q, \texttt{true})$;

$$m_p(\sigma) = \texttt{call-}m(A, M, Q, \texttt{false}) \ ,$$

if $p = (k_0, 0) \ldots (k_n, 0)$ and the $k_0 \ldots k_n$th occurrence of a stimulus complying to m is $\texttt{call-}m(A, M, Q, \texttt{false})$;

$$m_p(\sigma) = \texttt{return}(M, A, Q, \texttt{false}) \ ,$$

if $p = (k_0, 0) \ldots (k_n, 0)$ and the $k_0 \ldots k_n$th occurrence of a stimulus complying to m is $\texttt{return}(M, A, Q, \texttt{false})$.

Every action a adhering to a message m is provided with an initial state $\lambda_a^p : \Sigma \to \Lambda_a$ that may depend on the system state:

$$\begin{aligned} \lambda_a^p(\sigma) = {} & < \texttt{new}(\texttt{Action-}m, N) : \texttt{Action-}m \mid \texttt{ctxt} = G, \\ & \texttt{wait = false, target} = [\![e]\!], \texttt{cnt} = Q.0, \overrightarrow{rc = [\![r]\!]_{\mathrm{i}}} > \ , \end{aligned}$$

if $< H : \texttt{ProtoObject} \mid \texttt{class} = \texttt{Action-}m, \texttt{cnt} = N >$, $< O : C \mid \overrightarrow{b = V} >$, and $< G : \texttt{Activation-}m' \mid [\texttt{obj} = O,] \texttt{cnt} = Q, \overrightarrow{l = L} >$ are part of σ and $p = (k_0, l_0) \ldots (k_n, l_n)$ and $Q = k_0 \ldots k_n$; otherwise $\lambda_a^p(\sigma)$ may be chosen arbitrarily.

The transition relation $\rightarrow_a^\sigma$ of action a and system state σ is defined as follows:

$$< A : \texttt{Action-}m \mid \overrightarrow{b = V} > \rightarrow_a^\sigma \downarrow ,$$

if $< A : \texttt{Action-}m \mid \overrightarrow{b = V} >$ is part of σ and if the attributes `target` and `wait` of A are `empty` and `false`, resp., in σ:

$$< A : \texttt{Action-}m \mid \overrightarrow{b = V} > \rightarrow_a^\sigma < A : \texttt{Action-}m \mid \overrightarrow{b = V'} > ,$$

if $< A : \texttt{Action-}m \mid \overrightarrow{b = V} >$ is part of σ and if there is a $\sigma' \in \Sigma$ such that $\sigma \xrightarrow{\texttt{do-}m_i} \sigma'$ and $< A : \texttt{Action-}m \mid \overrightarrow{b = V'} >$ is part of σ' and $\overrightarrow{V} \neq \overrightarrow{V'}$, and if the attribute `target` of `A` is not `empty` in σ', but attribute `cnt` of A is the same in σ and σ';

$$< A : \texttt{Action-}m \mid \overrightarrow{b = V} > \rightarrow_a^\sigma (< A : \texttt{Action-}m \mid \overrightarrow{b = V'} >, \{\texttt{call-}m(A,\ M,\ Q,\ \texttt{false})\}) ,$$

if $< A : \texttt{Action-}m \mid \overrightarrow{b = V} >$ is part of σ and if there is a $\sigma' \in \Sigma$ such that $\sigma \xrightarrow{\texttt{do-}m_i} \sigma'$ and $< A : \texttt{Action-}m \mid \overrightarrow{b = V'} >$ and $\texttt{call-}m(A,\ M,\ Q,\ \texttt{false})$ are part of σ', and attribute `cnt` of A in σ' is $Q + 1$;

$$< A : \texttt{Action-}m \mid \overrightarrow{b = V} > \rightarrow_a^\sigma (< A : \texttt{Action-}m \mid \overrightarrow{b = V'} >, \{\texttt{return}(M,\ A,\ Q,\ \texttt{false})\}) ,$$

if $< A : \texttt{Action-}m \mid \overrightarrow{b = V} >$ is part of σ and if there is a $\sigma' \in \Sigma$ such that $\sigma \xrightarrow{\texttt{do-}m_i} \sigma'$ and $< A : \texttt{Action-}m \mid \overrightarrow{b = V'} >$ and $\texttt{return}(M,\ A,\ Q,\ \texttt{false})$ are part of σ', and attribute `cnt` of A in σ' is $Q + 1$; finally,

$$\rightarrow_a^\sigma = \emptyset ,$$

if none of the conditions above is satisfied.

With these definitions, let $\sigma_0 \in \Sigma$ be a start configuration as described in Sect. 3.2 and let $\sigma_0 \xrightarrow{l_0} \sigma_1 \xrightarrow{l_1} \ldots$ be a fair one-step run of Ξ. First of all, note that all the configurations σ_i are indeed in Σ since each rule preserves the unique numbering. Furthermore, a fair one-step run in particular implies that all actions eventually terminate, since their `target` attributes represent finite sets. We have to show that $(\sigma_k)_{k \in \mathcal{N}} \models \Phi$ where Φ is the conjunction of all instances of the temporal formulae (5.1–5.13) in Table 5.1 according to the interaction I:

(1) Actions can only be activated by a `start-`m rule which obviously creates actions in their initial state.

(2) Only the `do-`m and the `receive-`m rules affect an already existing action. If $a_p(\sigma_{k+1}) = (i', W', \lambda')$ differs from $a_p(\sigma_k) = (i, W, \lambda)$ for some $p \in (\mathcal{N} \times \mathcal{N})^+$ and some $k \in \mathcal{N}$, then the `wait` attribute changes from `true` to `false` and thus $a_p(\sigma_k) = (i, \{0\}, \lambda)$ and $a_p(\sigma_{k+1}) = (i, \emptyset, \lambda)$ holds; or the action terminated, i.e. `target` is `empty` and `wait` is false, and thus $a_p(\sigma_{k+1}) = \downarrow$ and $\lambda \downarrow_a^\sigma$ and $W = \emptyset$; or the `target` attribute is changed but no message is sent and thus $W = \emptyset$; or a message μ_0 is sent and thus $W = \emptyset$, $\alpha(m_{p.(i,0)})(\sigma_{k+1}) = \mu_0$, and $m_{p.(i,j)}(\sigma_{k+1}) = \perp$ for all $j > 0$.

(3) No rule applies to a terminated action.

(4) Messages can only be created by a `do-`m rule.

(5) Messages are not changed until received with a `receive-`m rule.

(6) Messages will be eventually received since the run is fair.

(7) Whenever an action is created by `start-`m, all immediate predecessors of its message have terminated.

(8) Actions can only be created in an activation context by a `start-`m rule; an activation context can only exist if a suitable message has been received by a `receive-`m rule.

(9) On receipt of a message, an activation context is created by a `receive-`m rule; all actions that have to be activated inside this activation context will be eventually created, since the run is fair.

(10) While waiting for a return message, an action cannot change; for each action at most one return message is awaited.

(11) No action exists in a start configuration.

(12) No action exists in a start configuration.

(13) All actions of messages that have no activator will be created eventually, since the run is fair.

Summing up, we have shown that a rewrite theory generated from a collaboration C according to Sect. 3.2 indeed is a model of the temporal logic formalisation in Sect. 1.3 describing the formal semantics of C's interaction.

Conclusions and Future Work

We generated rewrite theories from UML collaborations in the object-oriented algebraic specification language CafeOBJ, thus providing an executable model for collaborations and adding a simulation feature to UML. The context of a collaboration is reflected by a cast relation between instances and rôles; instances themselves are represented by objects. Interactions give rise to computational rules on configurations of objects and messages, representing the state of a collaboration's execution. This state consists of instances, stimuli, actions, and activation contexts. Rewrite clauses restrict the execution sequences of actions and the sending and receiving of stimuli according to the given interaction. Finally, we proved the generated rewrite theory correct with respect to a formal semantics of interactions in linear temporal logic.

The translation procedure can, in fact, be easily automated and we have implemented a simple prototype. This implementation has been carried out in Maude, using the reflective capabilities of the language [4]. The automatic generation of rewrite theories from

collaborations may hence also be conceived as a means to provide a formal specification for requirements put down in a collaboration, that may be refined later on a formal basis.

However, the coherence of different interactions in one collaboration combined by shared rôles or of different collaborations combined by shared actions has remained open. Furthermore, the translation sketched here has certainly to be extended to a richer subset of UML, giving rise to new questions of composition. Finally, target languages different from CafeOBJ may be taken into consideration.

REFERENCES

1. João Araújo. Formalizing Sequence Diagrams. In Luis F. Andrade, Ana Moreira, Akash R. Deshpande, and Stuart Kent, editors, *Proc. Wsh. Formalizing UML. Why? How?*, Vancouver, 1998.
2. Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesely, Reading, Mass., etc., 1998.
3. Peter Borovansky, Claude Kirchner, Hélène Kirchner, Pierre-Etienne Moreau, and Marian Vittek. ELAN: A Logical Framework Based on Computational Systems. In José Meseguer, editor, *Proc. 1st Int. Wsh. Rewriting Logic and Its Applications*, volume 4 of *Electr. Notes Theo. Comp. Sci.*, pages 35–50. Elsevier, 1996.
4. Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and José Quesada. Maude: Specification and Programming in Rewriting Logic. Manual, Computer Science Laboratory, SRI, 1999. `http://maude.csl.sri.com/manual`.
5. Razvan Diaconescu and Kokichi Futatsugi. *CafeOBJ Report — The Language, Proof Techniques, and Methodologies for Object-Oriented Algebraic Specification*, volume 6 of *AMAST Series in Computing*. World Scientific, Singapore, 1998.
6. Thomas Gehrke, Ursula Goltz, and Heike Wehrheim. The Dynamic Models of UML: Towards a Semantics and Its Application in the Development Process. Technical Report 11/98, Universität Hildesheim, 1998.
7. Alexander Knapp. A Formal Semantics for UML Interactions. In Robert B. France and Bernhard Rumpe, editors, *Proc. 2nd Int. Conf. UML*, volume 1723 of *Lect. Notes Comp. Sci.*, pages 116–130. Springer, Berlin, 1999.
8. Philippe Leblanc and Nicholas Dervaux. OMT/UML and SDL Based Techniques and Tools for Real-Time Development. In *Proc. Real-Time Systems 1998*. Teknea, Toulouse, 1998.
9. Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive and Concurrent Systems. Vol. 1: Specification*. Springer, New York–etc., 1992.
10. José Meseguer. A Logical Theory of Concurrent Objects and Its Realization in The Maude Language. In Gul A. Agha, Peter Wegner, and Akinori Yonezawa, editors, *Research Directions in Concurrent Object-Oriented Programming*, pages 314–389. MIT Press, Cambridge, Mass.–London, 1991.
11. Object Management Group. Unified Modeling Language Specification, Version 1.3. Technical report, Object Management Group, 1999. `http://www.omg.org/docs/ad/99-06-08.zip`.
12. Gunnar Övergaard. A Formal Approach to Collaborations in the Unified Modeling

Language. In Robert B. France and Bernhard Rumpe, editors, *Proc. 2nd Int. Conf. UML*, volume 1723 of *Lect. Notes Comp. Sci.*, pages 99–115. Springer, Berlin, 1999.

A. UML Meta-Model

```
module DATATYPES {
  protecting (IDENT)
  protecting (TERM)    — term representation using [...]
  protecting (NAT)     — natural numbers with infinity

  [ Name ]
  [ Ident < Name ]

  [ ParameterDirectionKind ]
  ops pdk-in, pdk-out, pdk-inout, pdk-return : -> ParameterDirectionKind

  [ Multiplicity ]
  op [_..._] : Nat* Nat* -> Multiplicity
}

module UML {
  protecting (DATATYPES)
  protecting (SET[ObjectId])
  protecting (LIST[ObjectId])

  class Element { }
  class ModelElement < Element { name : Name }
  class Namespace < ModelElement { }
  class GeneralizableElement < ModelElement { }

  class Classifier < Namespace, GeneralizableElement {
    generalization : Set[ObjectId]  feature : List[ObjectId] }
  class Class < Classifier { isActive : Bool }
  class Actor < Classifier { }
  class Feature < ModelElement { }
  class StructuralFeature < Feature { type : ObjectId }
  class BehaviouralFeature < Feature { parameter : List[ObjectId] }
  class Attribute < StructuralFeature { }
  class Operation < BehaviouralFeature { method : Set[ObjectId] }
  class Method < BehaviouralFeature { specification : ObjectId }
  class Parameter < ModelElement {
    type : ObjectId, kind : ParameterDirectionKind }

  class Relationship < ModelElement { }
  class Association < GeneralizableElement, Relationship {
    connection : List[ObjectId] }
  class AssociationEnd < ModelElement {
    isNavigable : Bool  isOrdered : Bool
    multiplicity : Multiplicity  type : ObjectId }
  class LocalAssociationEnd < AssociationEnd { local : ObjectId }

  class Collaboration < Namespace {
    representedOperation : ObjectId representedClassifier : ObjectId
    ownedElement : Set[ObjectId]  interaction : ObjectId }
  class ClassifierRole < Classifier {
    availableFeature : List[ObjectId]
    multiplicity : Multiplicity  base : ObjectId }
  class AssociationRole < Association { base : ObjectId }
```

```
  class AssociationEndRole < AssociationEnd { base : ObjectId }
  class LocalAssociationEndRole <
          AssociationEndRole, LocalAssociationEnd { }
  class Interaction < ModelElement {
    context : ObjectId  message : Set[ObjectId] }
  class Message < ModelElement {
    predecessor : Set[ObjectId]  activator : ObjectId
    action : ObjectId  sender : ObjectId  receiver : ObjectId }
  class Action < ModelElement {
    recurrence : Term  target : Term,
    isAsynchronous : Bool  actualArgument : List[ObjectId] }
  class CallAction < Action { operation : ObjectId }
  class ReturnAction < Action { }
  class Argument < ModelElement { expression : Term }
}
```

B. Sample UML Model

```
  < U : Actor | name = U, generalization = empty, feature = nil >
  < A : Class | name = A, generalization = empty,
                feature = (A-k :: A-l :: nil), isActive = false > ...
  < C : Class | name = C, generalization = empty,
                feature = (C-n :: nil), isActive = false > ...

  < UA : Association | name = UA, connection = (UA-u :: UA-a :: nil) > ...

  < UA-u : AssociationEnd |
      name = UA-u, isNavigable = true, isOrdered = false,
      multiplicity = [1 .. 1], type = U >
  < UA-a : AssociationEnd |
      name = UA-a, isNavigable = true, isOrdered = false,
      multiplicity = [1 .. 1], type = A >

  < A-k : Operation | name = A-k, parameter = nil, method = empty >
  < A-l : Operation | name = A-l, parameter = nil, method = empty > ...
  < C-n : Operation |
      name = C-n, parameter = (C-n-y :: C-n-ret :: nil),
      method = empty > ...

  < C-n-y : Parameter | name = C-n-y, type = D-List, kind = pdk-in >
  < C-n-ret : Parameter | name = C-n-ret, type = D, kind = pdk-return >

  < CC : Collaboration |
      name = CC, represented = null, interaction = I,
      ownedElement = { U-Role, P, Q, R, S, UP, PQ, PR, PS } > ...

  < P : ClassifierRole |
      name = P, generalization = empty, feature = nil,
      isActive = false, multiplicity = [1 .. 1],
      availableFeature = (A-k :: A-l :: nil), base = A > ...
  < S : ClassifierRole |
      name = S, generalization = empty, feature = nil,
      isActive = false, multiplicity = [0 .. infty],
      availableFeature = (D-o :: nil), base = D > ...

  < PQ : AssociationRole |
      name = PB, connection = (PQ-a :: PQ-b :: nil),
```

```
    multiplicity = [1 .. 1], base = AB > ...
< PS : AssociationRole |
    name = PD, connection = (PS-a :: PS-d :: nil),
    multiplicity = [0 .. infty], base = AD >
< PSL : AssociationRole |
    name = PSL, connection = (PSL-a :: PSL-d :: nil),
    multiplicity = [1 .. 1], base = AD > ...

< PQ-a : AssociationEndRole |
    name = PQ-a, isNavigable = true, isOrdered = false,
    multiplicity = [1 .. 1], type = P, base = AB-a > ...
< PSL-d : LocalAssociationEndRole |
    name = r, isNavigable = true, isOrdered = false,
    multiplicity = [1 .. 1], type = S, base = AD-d, local = m1 >

< I : Interaction | context = C, message = { m1, m2, m1-1A,
    m1-1B, r1-1B, m1-1C, r1-1C, m2-1, r2-1 } >
< m1 : Message |
    name = m1, interaction = I, predecessor = empty,
    activator = null, action = c-a-k, sender = U-Role, receiver = P > ...
< m1-1A : Message |
    name = m1-1A, interaction = I, predecessor = empty,
    activator = m1, action = c-b-m, sender = P, receiver = Q >
< m1-1B : Message |
    name = m1-1B, interaction = I, predecessor = empty,
    activator = m1, action = c-c-n, sender = P, receiver = R >
< r1-1B : Message |
    name = r1-1B, interaction = I, predecessor = empty,
    activator = m1-1B, action = r-c-n, sender = R, receiver = P > ...

< c-a-k : CallAction |
    name = c-a-k, recurrence = 1, target = (makeSet[1, UP-a]),
    isAsynchronous = true, actualArgument = nil, operation = A-k > ...
< c-b-m : CallAction |
    name = c-b-m, recurrence = (iterate[i, 0, length[PS-d]]),
    target = (makeSet[length[PS-d], PQ-b]),
    isAsynchronous = true, actualArgument = (c-b-m-1 :: nil),
    operation = B-m >
< c-c-n : CallAction |
    name = c-c-n, recurrence = 1, target = (makeSet[1, PR-c]),
    isAsynchronous = false, actualArgument = (c-c-n-1 :: nil),
    operation : C-n > ...
< r-c-n : ReturnAction |
    name = r-c-n, recurrence = 1, target = (makeSet[0, null]),
    isAsynchronous = false, actualArgument = (r-c-n-1 :: nil) > ...

< c-b-m-1 : Argument | name = c-b-m-1, value = (ith[PS-d, i]) >
< c-c-n-1 : Argument | name = c-c-n-1, value = PS-d >
< r-c-n-1 : Argument | name = r, value = (hd[C-n-y]) >
```

Chapter 6

CASL for CafeOBJ Users

Peter D. Mosses[a]

[a]BRICS & Department of Computer Science, University of Aarhus,
Ny Munkegade bldg. 540, DK-8000 Aarhus C, Denmark
Home page: http://www.brics.dk/~pdm

CASL is an expressive language for the algebraic specification of software requirements, design, and architecture. It has been developed by an open collaborative effort called CoFI (Common Framework Initiative for algebraic specification and development). CASL combines the best features of many previous main-stream algebraic specification languages, and it should provide a focus for future research and development in the use of algebraic techniques, as well facilitating interoperability of existing and future tools.

This paper presents CASL for users of the CafeOBJ framework, focussing on the relationship between the two languages. It first considers those constructs of CafeOBJ that have direct counterparts in CASL, and then (briefly) those that do not. It also motivates various CASL constructs that are not provided by CafeOBJ. Finally, it gives a concise overview of CASL, and illustrates how some CafeOBJ specifications may be expressed in CASL.

1. Introduction

CASL is an expressive language for the algebraic specification of software requirements, design, and architecture. It has been developed by an open collaborative effort called CoFI (Common Framework Initiative for algebraic specification and development).

This paper presents CASL for users of the CafeOBJ framework.

▷ CASL is intended as the main language of a coherent family of languages.

Vital for the support for CoFI in the algebraic specification community is the coverage of concepts of many previous specification languages. How could this be achieved, without creating a complicated monster of a language? And how to avoid interminable conflicts with those needing a simpler language for use with prototyping and verification tools?

By providing not merely a single CASL language but a coherent language family, CoFI allows the conflicting demands to be resolved, accommodating advanced as well as simpler languages. At the same time, this family is given structure by being organized largely as restrictions and extensions of CASL.

Restrictions of CASL [13,14] correspond closely to languages used with existing tools for rapid prototyping, verification, term rewriting, etc. Extensions to CASL are to support various programming paradigms, e.g. object-oriented, higher-order [11], reactive.

▷ A specification framework is more than just a language.

Apart from the CASL family of languages, the common framework is also to provide tool support, development methodologies, training materials, libraries, reference manuals, formal semantics and proof systems, and conversion to and from other specification languages.

▷ CASL is competitive in expressiveness with many previous algebraic specification languages.

The choice of concepts and constructs for CASL was a matter of finding a suitable balance between advanced and simpler languages, taking into account its intended applicability: for specifying the functional requirements and design of conventional software packages as abstract data types.

The design of CASL is based on a critical selection of the concepts and constructs found in previous algebraic specification frameworks. The main novelty of CASL lies in its particular *combination* of concepts and constructs, rather than in the latter *per se*.

▷ The CASL design has been tentatively approved by IFIP WG 1.3.

The design proposal for CASL [7] was submitted to IFIP Working Group 1.3 (Foundations of System Specification) in May 1997, and tentatively approved at the IFIP WG 1.3 meeting in Tarquinia, June 1997 (subject to reconsideration of some particular points [6] and the development of a satisfactory concrete syntax, both of which have now been completed [8]). The formal semantic description of CASL has been completed [9]. Tools and methodology for CASL are being developed.

▷ CoFI is open to contributions and influence from all those working with algebraic specifications.

The design of CASL has been the main responsibility of the CoFI Language Design task group, coordinated by Bernd Krieg-Brückner. At any time during the three years it took to design CASL, there were 10–20 active participants in this task group—most of them with previous experience of designing algebraic specification frameworks. The work of the CoFI task groups on Methodology, Semantics, and Tools influenced and provided essential feedback to the language design. Numerous study notes were written on various aspects of language design, and discussed at working and plenary meetings. The study notes and various drafts of the design summary were made available electronically, and discussed on the language design mailing list (`cofi-language@brics.dk`).

The openness of the CASL design effort should have removed the risk of bias towards constructs favoured only by some particular 'school' of algebraic specification. It is hoped that CASL incorporates just those features for which there will be a *wide consensus* regarding their appropriateness, and that the common framework will indeed be able to subsume many previous frameworks and be seen as an attractive basis for future development and research—with high potential for strong collaboration.

So much for the background of CASL.

▷ Readers of this paper are assumed to be familiar with the CafeOBJ language and system.

For an introduction to CafeOBJ, see the CafeOBJ Report [10].

▷ The aims of CASL are somewhat different from those of CafeOBJ.

From The CafeOBJ Official Home Page[1] (April 2000):

> *CafeOBJ is a new generation algebraic specification and programming language. As a direct successor of OBJ, it inherits all its features (flexible mix-fix syntax, powerful typing system with sub-types, and sophisticated module composition system featuring various kinds of imports, parameterized modules, views for instantiating the parameters, module expressions, etc.) but it also implements new paradigms such as rewriting logic and hidden algebra, as well as their combination.*

In fact CASL too enjoys flexible mix-fix syntax, a powerful typing system with sub-types, and a module composition system featuring imports, parameterized modules, views for instantiating the parameters, module expressions, etc. Many of these features of CASL were directly inspired by OBJ3, and their inclusion reflects that they are indeed seen as part of the well-established, main-stream, traditional algebraic specification practice.

▷ CASL does not provide direct support for the new paradigms of rewriting logic and hidden algebra.

The omission of these new paradigms is partly because when CASL was being designed, they were indeed rather new—and are still perhaps not generally regarded as belonging to the main-stream of algebraic specification—partly because they were felt to be somewhat too advanced for inclusion in a general-purpose specification language. The design of CASL reflects the state-of-the-art of the early 1990's [2], and leaves more recent advances to be incorporated in extensions of CASL (which are also to provide features oriented towards particular programming paradigms, such as object orientation and concurrency).

As a consequence, CASL is directly comparable only to the equational specification fragment of CafeOBJ. The design of such a specification language is, however, still a non-trivial task, and it is quite interesting to compare the design of CASL with that of equational CafeOBJ.

▷ CASL specifications need not be executable.

From the CafeOBJ Report [10]:

> *[Equational specification and programming] is inherited from OBJ [...] and constitutes the basis of the language, the other features being built on top of it. As with OBJ, CafeOBJ is* executable, *which gives an elegant declarative way of functional programming, often referred [to] as* algebraic programming.

[1] http://www.ldl.jaist.ac.jp/cafeobj/

Executability is an important concern also in the rewriting logic and hidden algebra paradigms, both for testing specifications and for reasoning about them.

CASL is intended primarily for expressing requirements and design decisions during software development. The constructs of CASL have been selected on the grounds of their expressiveness and semantic simplicity—but without any regard to their executability. Of course, it is important for the authors and readers of specifications to be able to investigate the consequences of axioms; for CASL specifications, this may require the use of interactive theorem-provers, in general (CASL interfaces to the Isabelle and INKA systems are already available). Nevertheless, automated rapid prototyping of some CASL specifications should still be possible. Various executable sub-languages of CASL are in any case provided [12,14].

When requirements are expressed independently of executability concerns, specifications may gain clarity and keep closer to informal statements of requirements. As Amir Pnueli put it in his invited talk at ETAPS'98: "requirement specifications and programs are two different views that should be checked (and debugged) against each other".

With the stated differences of aims, it is only to be expected that there are some differences between the design of CafeOBJ and that of CASL. The purpose of this paper is to explain both similarities and differences—addressing primarily readers who are already familiar with CafeOBJ. Please note that both languages seem to reflect coherent designs, and that individual design choices cannot be changed without reconsideration of many other interacting issues, in general. Thus no critique of the overall CafeOBJ design is implied when some advantages of particular CASL features are pointed out (or *vice versa*).

▷ A proper comparison of the traditional algebraic approach with the new paradigms incorporated in CafeOBJ is out of the scope of this paper.

Such a comparison would have to take into account many methodological aspects of software specification and development, including case studies. The focus of this paper is on presenting CASL and explaining how its language constructs relate to those of CafeOBJ. The many novel features of CafeOBJ that have no direct counterpart in CASL are mentioned only very briefly.

Plan

First we consider the intersection of CafeOBJ and CASL: those concepts and constructs that are common to both languages. For each such CafeOBJ construct, we see how it might be expressed concretely in CASL, and consider any significant differences in the details. Then we mention the remaining constructs of CafeOBJ: those that cannot (straightforwardly) be expressed in CASL. After that, we list those constructs of CASL that cannot (straightforwardly) be expressed in CafeOBJ, and motivate their inclusion in CASL. We finish the presentation of CASL by summarizing its constructs, and by giving a few simple examples of CASL specifications.

▷ All the main points in this paper are displayed like this.

The paragraphs following each point provide details and supplementary explanation. To get a quick overview, simply read the main points and skip the intervening text. It is

hoped that the display of the main points does not unduly hinder a continuous reading of the full text. (This style of presentation was inspired by a book of Alexander [1].)

2. The Common Features: CafeOBJ ∩ CASL

▷ CafeOBJ and CASL have a large number of features in common.

In this section we consider the intersection of CafeOBJ and CASL: those concepts and constructs that are common to both languages. For each such CafeOBJ construct, we see how it may be expressed in CASL. Translation between specifications involving just these constructs could probably be automated without much difficulty (although to prove that the translation is somehow semantics-preserving might not be so easy).

The high degree of overlap between the equational fragment of CafeOBJ and CASL reflects the extent to which their designs have been influenced by previous main-stream algebraic specification frameworks, of which OBJ3 is a good example. Note however that CafeOBJ was designed deliberately as a successor to OBJ3, whereas the CASL design was intended to be unbiased towards any particular previous framework (and it is as close to LSL, the Larch Shared Language, as it is to OBJ3).

Both CafeOBJ and CASL distinguish between basic and structured specifications. Let us focus first on their main constructs, deferring the various abbreviatory 'convenience features' to the end of the section.

It may be helpful to see some comparable examples straight away. The following simple examples of equational specifications in CafeOBJ are taken from the CafeOBJ Report [10, pages 18–19] (where they are used to illustrate the CafeOBJ treatment of partial functions by sort membership predicates):

```
mod* GRAPH {
  [ Node Edge ]

  ops (s_) (t_) : Edge -> Node
}

mod! PATH (G :: GRAPH) {
  [ Edge < Path ]

  op _;_ : ?Path ?Path -> ?Path (assoc)
  ops (s_) (t_) : Path -> Node

  var E : Edge
  var EP : ?Path

  ceq (E ; EP) : Path = true
       if (EP : Path) and (s EP) == (t E) .
  ceq s(E ; EP) = s(E)  if (E ; EP) : Path .
  ceq t(E ; EP) = t(EP) if (E ; EP) : Path .
}
```

Here is how the same examples[2] could be expressed in CASL (illustrating its straightforward treatment of partiality without involving error sorts):

spec GRAPH =
  sorts $Node, Edge$
  ops $s\_, t\_ : Edge \to Node$

spec PATH[GRAPH] = **free** {
  sorts $Edge < Path$
  op $\_ :: \_ : Path \times Path \to ? Path, assoc;$
    $s\_, t\_ : Path \to Node$
  vars $E : Edge; EP : Path$
  • $def\ E :: EP\ if\ s\ EP = t\ E$
  • $s(E :: EP) = s(E)\ if\ def\ E :: EP$
  • $t(E :: EP) = t(EP)\ if\ def\ E :: EP$ }

2.1. Basic Specifications

▷ Both languages allow declarations and axioms to be interleaved.

Basic specifications consist of declarations and axioms, written in any order such that symbols are declared before they are used.

▷ A basic specification determines a signature and a class of models.

Both CafeOBJ and CASL define the semantics of a specification in terms of model classes (rather than theories). Moreover, both provide so-called institutions for basic specifications, connecting the notions of declarations, axioms, models, and satisfaction. Both also provide the means to restrict the models of a specification to initial models.

Declarations

▷ Declarations of sorts and subsorts are similar in CafeOBJ and CASL.

The general form of a declaration of one or more sorts, together with subsort inclusions, is in CafeOBJ:

$$[\ S_{11}\ \ldots\ S_{1n_1} < \ldots < \ S_{m1}\ \ldots\ S_{mn_m}\ ]$$

and in CASL:

$$\textbf{sorts}\ S_{11}, \ldots, S_{1n_1} < S'_1; \quad \ldots; \quad S_{m1}, \ldots, S_{mn_m} < S'_m$$

[2] In fact the semantics is not quite the same: CASL does not allow models with empty carriers, so neither the empty graph, nor discrete (edge-less) graphs are models of the specification GRAPH. This discrepancy could be remedied in various ways, not relevant to the illustrative purpose of these examples.

Thus a little extra conciseness is obtainable in CafeOBJ (when the same sort is declared to be a subsort of more than one supersort).

▷ The interpretation of subsort declarations is slightly different.

CafeOBJ interprets subsort declarations as *set inclusions* between the corresponding carrier sets. The CASL interpretation is a bit more general, allowing (1-1) embeddings as well as inclusions. Such embeddings are especially relevant in the context of software design and implementation, where the models considered are not necessarily term models, since they permit more efficient representations to be used for the restricted sets of values in subsorts. For instance, in models of a CASL specification of numbers with subsorts, the representation of values in the sort of integers may be different from their embeddings into a sort of reals or rationals. (Overloaded operations are required to commute with subsort embeddings, so the satisfaction of formulae such as $2 + 2 = 4$ is independent of the subsort embeddings involved.) Moreover, the subsort relation in CASL is in general only a pre-order, rather than a partial order, and it is possible to declare that different sorts are to be isomorphic.

▷ Declarations of total operations are similar.

A declaration of an ordinary, total operation (or constant) is in CafeOBJ:

op o : s_1 ... s_n -> s

In CASL it is:

op $o : s_1 \times \ldots \times s_n \to s$

or just **op** $o : s$ when the operation is just a constant.

By the way, the non-ASCII mathematical symbols shown in CASL specifications are a feature of the CASL display format, and CASL specifications have to be input in ASCII (or ISO Latin-1). For instance, '$\times$' may be input as '*', or directly in ISO-Latin-1, and '$\to$' is always input as '->'.

▷ Declarations of partial operations look quite similar, but have different semantics.

CafeOBJ does not directly support partial operations, but rather handles them indirectly via error sorts and a sort membership predicate. A declaration of a partial operation is thus written:

op o : s_1 ... s_n -> ?s

where '?s' is an error sort (implicitly declared as a supersort of s when s is maximal in a connected component of sorts). An application of the operation is well-formed provided that the sort of each argument term is in the same connected component as the declared sort of that argument (as in Membership Equational Logic). The definedness of the value of a term corresponds to whether it belongs to a non-error sort, and can be tested (or asserted) using the sort membership predicate, written '$T : s$'. Equations may (but need not) hold when the values of the two terms are both error values.

CASL supports partial operations directly, without implicit error sorts (cf. the example given at the beginning of Sect. 2). A declaration of a partial operation is written:

op $o : s_1 \times \cdots \times s_n \to ? \; s$

Note that in CASL, the '?' is part of the notation for declaring the profile (argument and result sorts) of operations, and '?s' is not even a valid sort name. Moreover, the result sort is not required to be maximal among the declared sorts. The definedness of the value of a term is expressed by a special atomic formula, written '*def* T'. Ordinary equations hold when both terms have equal defined values, and (always) when both values are undefined.

One noticeable difference between the two treatments of partiality is in the well-formedness of terms: in CafeOBJ, the check for well-formedness is as if each sort were to be replaced by the error supersort of its connected component of the sort hierarchy; in CASL, the check for well-formedness insists that the sorts of argument terms are subsorts of the corresponding argument sorts declared for the operation (the difference disappears when operations in CASL are declared as taking arguments in maximal supersorts).

Another difference is that in CASL, the undefinedness of any argument in an application of an operation implies that the result of the application is undefined too; moreover, a predicate never holds when any argument is undefined. In CafeOBJ, applying an operation to error values may result in a non-error value. To get the same effect in CASL one would have to declare supersorts and error values explicitly.

▷ Also declarations of predicates in CASL and CafeOBJ look similar, but here too there are some significant differences.

A declaration of a predicate is in CafeOBJ:

pred p : s_1 ... s_n

and in CASL:

pred $p : s_1 \times \cdots \times s_n$

Predicates in CafeOBJ are, however, merely syntactic sugar for operations to the built-in sort `Bool`. To express that a predicate holds in an axiom (or not) the value of the application has to be compared to the constant `true` (or `false`). Predicates may also be used in `Bool`-sorted arguments to ordinary operations. Note that there is a distinction between the ordinary equality '`=`' and the corresponding predicate '`==`', in that the former is an atomic formula and cannot be used at all inside terms.

In CASL, predicates are kept separate from operations. An application of a predicate to argument terms is itself an atomic formula, and does not need comparing with true or false—in fact in CASL there is no built-in sort of Boolean values at all (a standard Boolean specification is however available for use if needed, for instance in sub-languages that do not allow user-declared predicates).

Regarding semantics, the CASL treatment entails that in initial models, predicates fail to hold by default (just like equations), so it is sufficient to specify only the cases where they are supposed to hold. In CafeOBJ, however, to get the expected operational semantics and initial model class, it is necessary to specify also the cases where the result is intended to be false. In fact in the `PATH` example given in CafeOBJ at the beginning of Sect. 2, the implicit constraint that the sort `Bool` (declared by a built-in specification that is inherited by every CafeOBJ specification, and used in the representation of sort

membership predicates) has only two elements prevents the existence of initial models, making (most instantiations of) the given specification inconsistent.[3]

The fundamental reason for this semantic difference is that the values `true` and `false` are treated uniformly by model homomorphisms in CafeOBJ models, whereas in CASL, homomorphisms preserve only the holding of predicates, not their non-holding. (The CASL semantics of predicates is equivalent to representing them as *partial* operations to a sort constrained to have just a single element, the definedness of the operation corresponding to the holding of the predicate.)

▷ Both languages allow overloading.

In CafeOBJ, overloading is allowed provided that signatures are 'sensible':

> *If the same operation name is declared with argument sorts* $s_1, \ldots, s_n$ *and result sort* s *and also with argument sorts* $s'_1, \ldots, s'_n$ *and result sort* s'*, then* s *and* s' *are in the same connected component iff for each* i*,* s_i *and* s'_i *are in the same connected component.*

Although this condition appears to be an improvement on several previous conditions proposed for order-sorted signatures, it prohibits declaring the same *predicate* symbol for unconnected argument sorts (in the same module), e.g. '`_<=_`' for both numbers and strings, since predicates are simply operations of sort `Bool`. Moreover, it appears that the implicitly-declared equality predicate '`_==_`' *always* leads to non-sensible signatures (even when there is only one declared sort—recalling that error sorts are implicitly declared). A relatively minor further point is that constants cannot be overloaded between different connected components, whereas it could be convenient to use the same symbol for the nil list in different sorts of lists, for instance. Finally, in connection with structured specifications, there is the problem that the restriction to sensible signatures might not be preserved by the structuring operations.

CASL, in contrast, does not impose any restrictions on overloading at all. The combination of overloading of operations with subsort inclusions between their argument and result sorts simply entails some implicit axioms, which ensure that terms have the same value whenever they are identical up to the commuting of overloaded operations with subsort inclusions. When axioms are well-formed, their satisfaction is insensitive to overloading resolution. (One might fear that such a lack of discipline would unduly complicate overloading resolution, but it can be implemented so that in the case the signature happens to be regular, the complexity is that same as that in OBJ3, and moreover the slow-down appears to be insignificant for the non-regular signatures that occur in practice. In any case, the authors of specifications have the possibility of indicating the intended sorts when using heavily-overloaded operations.)

[3] Also [10, Example 20] implicitly imports `BOOL` and exhibits the same problem. A referee pointed out that one may override the implicit importation of `Bool` in protecting mode in such examples by importing it explicitly in extending mode; however, the consequences of thus allowing 'junk' values in `Bool` in CafeOBJ are unclear.

Axioms

▷ Both languages allow equations.

In CafeOBJ equations are written 'T_1 `=` T_2', except in conditions, where they are written as applications of the equality predicate: 'T_1 `==` T_2'.

Also CASL has two kinds of equations, but here the difference is with respect to undefined values. An ordinary equation '$T_1 = T_2$' is interpreted as asserting that either both values are both defined and equal, or they are both undefined. An existential equation '$T_1 \stackrel{e}{=} T_2$' asserts that the values are both defined and equal. The two kinds of equations are actually inter-definable in CASL, but they are both provided as language constructs, since existential equations are more appropriate in conditions (one does not usually want coincidental undefinedness of two terms to have further consequences) and ordinary equations are particularly useful in inductive definitions of partial functions.

▷ Both languages allow sort membership assertions.

In CafeOBJ sort membership is a predicate, and the assertion that a term T has sort s is written 'T `:` s `= true`'.

In CASL, a sort membership assertion is an atomic formula, written '$T \in S$'.

▷ Both languages allow conditional formulae.

In CafeOBJ a conditional equation is written '`ceq` T_1 `=` T_2 `if` T', where T is a term of sort `Bool`, and may thus involve not only the equality and subsort membership predicates but also all the logical connectives that are defined on `Bool`.

In CASL, a conditional equation is a special case of a conditional formula, which may be written in two ways: '$F_1 \Rightarrow F_2$' (not to be confused with the CafeOBJ notation for rewriting transitions) or 'F_2 *if* F_1', where the F_i are arbitrary formulae.

2.2. Structured Specifications

Specifications may be structured both by module expressions and by the naming of (possibly parameterized) specification modules.

▷ There are some syntactic differences between CafeOBJ and CASL concerning module expressions and module bodies.

In CafeOBJ, module expressions are used mainly for specifying parameters of generic specifications and views, whereas in CASL they are also used as the bodies of named specifications. Moreover, although both languages allow references to named specifications in module expressions, CafeOBJ does not (according to [10, Appendix]) allow a reference to a named specification to be replaced by its body, nor the direct use of a basic specification in a module expression, in contrast to CASL.

Extensions

▷ Both languages allow extensions of named specifications.

In CafeOBJ, an extension is always specified by a named module, which refers to the named modules being extended as imports, along with declarations and axioms in its body:

`mod` SN ... { m_1 SN_1 ... m_n SN_n ... }

The mode m of each import is specified as `protecting`, `extending`, or `using`.

In CASL, an extension is itself a module expression 'SP_1 **then** SP_2'. The combination of extension with naming is written:

spec SN ... = SN_1 **and** ... **and** SN_n **then** ...

No explicit mode for the extension is specified (although one may use annotations, not affecting the algebraic semantics of specifications, to indicate properties such as the conservativeness of an extension).

▷ Both languages allow protecting extension of imports.

The strongest importation mode of CafeOBJ, '`protecting`', allows the expansion to add new sorts, operations, and predicates, but not to add new values to previous sorts, nor to equate old values. If the semantics of the imported specification is loose, however, a 'protecting' extension that requires new values, or the identification of old values, can still have models: namely, those models of the imported specification that *already* had such 'junk' or 'confusion'. This mode of importation corresponds to ordinary extension in CASL: 'SP_1 **then** SP_2'.

▷ Both languages allow loose and initial semantics.

In CafeOBJ the semantics of a module is specified by the keyword used when naming it: '`mod*`' gives loose semantics, '`mod!`' gives initial semantics.

In CASL, the initial semantics of a specification is obtained from the default loose semantics using the specification expression '**free** SP'. When used together with naming it is written:

spec SN ... = **free** ...

In both CafeOBJ and CASL, it may happen that the loose semantics of a specification is a non-empty class of models that has no initial model, in which case its initial semantics is the empty model class.

▷ Both languages allow free extension.

Free extension in CafeOBJ generalizes initial semantics to the case when the specification has imports, still being specified by use of the keyword '`mod!`' when naming the specification. CafeOBJ takes all models of the extension that are free extensions of *any* models of the imports (according to the somewhat informal semantics given in the CafeOBJ Report [10]).

Also in CASL, free extension is specified by a combination of the constructs used for expressing initial semantics and extension, being written 'SP_1 **then free** SP_2'. It denotes the class of models of the extension that are free extensions of *their own reducts* to SP_1.

In both languages, the possibility of combining loose semantics and free extensions allows specifications whose semantics is partly loose and partly initial (as with the use of so-called data constraints in Clear).

▷ Both languages allow union of specifications.

The CafeOBJ construct 'SP_1 `+` SP_2' is called a shared sum, and it is somewhat different from the union construct 'SP_1 **and** SP_2' in CASL. In CafeOBJ, a symbol declared by both SP_1 and SP_2 is regarded as the same symbol only when it has a unique origin in some imported module, in general; thus some kind of qualification of symbols by module names may be needed to disambiguate identical symbols of different origins.

In CASL, however, all common symbols of SP_1 and SP_2 are regarded as identifying the *same* entities (cf. union of traits in LSL). In fact the 'same name, same thing' philosophy is quite pervasive in CASL. (This does not prevent overloading, since operation symbols with different profiles are considered to be different names.)

Clearly, when specifications written by different people are united, there is a danger of unintentional clashes of names. But in CASL it is straightforward to hide auxiliary symbols that are introduced purely for internal use, leaving visible only the symbols that were originally *intended* to be specified. (N.B. The notion of hiding here is completely different from that found in the 'hidden algebra' approach.) Unintentional clashes of names in the *interface* of a module *need* to be resolved in any case, by renaming. Tools should be able to warn about clashes that might be unintentional (ignoring symbols that get indirectly imported from the same origin).

▷ Both languages allow translation of symbols declared by specifications.

A translation in CafeOBJ is written:

$$SP \texttt{ * \{ ..., sort } s_1 \texttt{ -> } s_2 \texttt{, ..., op } o_1 \texttt{ -> } o_2 \texttt{, ... \}}$$

and in CASL:

$$SP \textbf{ with } \ldots, s_1 \mapsto s_2, \ldots, o_1 \mapsto o_2, \ldots$$

(Keywords such as '**sort**' and '**op**' may be inserted here also in CASL, when desired.) Symbols that are to be left unchanged may be omitted in both languages.

Parameters

▷ Both languages allow parameterized modules.

A module with a parameter in CafeOBJ is written:

`mod` *SN* (*X*$_1$ `::` *m*$_1$ *SP*$_1$) { ... }

and in CASL:

spec *SN* [*SP*$_1$] = ...

In CASL, the parameter is simply a specification expression (often just a name, in practice) without any label or mode, simply indicating which part of the extension of the parameter *SP*$_1$ by the body *SP* is intended to vary.

▷ Both languages allow instantiation of parameters with module expressions.

In CafeOBJ an instantiation of parameter *X*$_1$ with a module expression *SP*$_2$ in a module named *SN* can be written '*SN*(*X*$_1$ `<= view to` *SP*$_2${*SM*$_1$})', where *SM*$_1$ is a symbol mapping (as in a translation). In simple cases the label *X*$_1$ can be omitted and the explicit view can be replaced by *SP*$_2$ itself, giving simply '*SN*(*SP*$_2$)'

In CASL a similar instantiation can be written '*SN*[*SP*$_2$ **fit** *SM*$_1$]', or just '*SN*[*SP*$_2$]' in simple cases.

The semantics of instantiation corresponds to a pushout construction in both languages.

▷ Both languages allow views to be named and used in instantiations.

The fitting between a particular parameter specification and argument specification may itself be named in CafeOBJ: '`view` *VN* `from` *SP*$_1$ `to` *SP*$_2$ { *SM*$_1$ }' and *VN* may then be used instead of the explicit view in instantiations.

Named views are defined similarly in CASL: '**view** *VN* **from** *SP*$_1$ **to** *SP*$_2$ = *SM*$_1$', whereafter '**view** *VN*' may then be used instead of '*SP*$_2$ **fit** *SM*$_1$'.

▷ Although both languages allow multiple parameters, their treatment of shared symbols is different.

The basic treatment of multiple parameters in CafeOBJ is non-sharing, where the same symbol occurring in different parameters is distinguished by the different labels of the parameters. However, there is also a construct '`share`(*SN*)' that allows unwanted duplication of the module *SN* (due to importation in different instantiated parameters) to be avoided.

In contrast, multiple parameters in CASL are regarded as parts of a single united parameter, thus symbols common to different parameters always share, and have to be instantiated uniformly. Multiple parameters that are intended to be independent should therefore always have distinct symbols. Thus to specify generic pairs with different sorts of components, one should declare the parameters as PAIR[**sort** *Elem1*][**sort** *Elem2*], rather than as PAIR[**sort** *Elem*][**sort** *Elem*].

The CASL 'same name, same thing' philosophy eliminates any need for 'dot notation', which is used in CafeOBJ to determine the parameter to which a symbol is supposed to refer.

2.3. Convenience Features

▷ Both languages allow attributes in operation declarations.

Both CafeOBJ and CASL provide the attributes of associativity, commutativity, idempotency, and unit/identity for binary operations.

▷ Both languages allow simultaneous declarations.

Several operations with the same profiles (and attributes) may be declared together, abbreviating a sequence of declarations.

▷ Both languages allow both global and local variable declarations.

Explicit variable declarations are global in CafeOBJ, whereas implicit declarations in terms are presumably local to the rest of the enclosing equation.

Variable declarations in CASL are global, abbreviating universal quantification over the declared variables in all the subsequent axioms of the enclosing basic specification—unless they are followed by a '•', in which case their scope is restricted to the following •-separated list of axioms.

▷ Both languages allow mixfix notation.

The usefulness of mixfix notation, generalizing infix, prefix, and postfix notation to allow arbitrary sequences of fixed tokens and arguments, is so obvious that the CafeOBJ Report [10] hardly bothers to mention it. The main difference in CASL is that place-holders are written with double underscores (leaving single underscores available for separating words in identifiers).

3. The Differences: CafeOBJ \ CASL

The main features of CafeOBJ that are missing from CASL include built-in support for hidden algebra, behavioural equivalence, and rewriting logic, as well as indications of import and parameter modes.

3.1. Basic Specifications

▷ CASL does not allow 'hidden' sorts or behavioural operations.

Sort symbols can be hidden in CASL, but the meaning is completely different from hidden sorts in CafeOBJ: in CASL, the sorts are simply removed from the declared signature (together with all operations whose profiles involve them).

Similarly, there is no way of distinguishing behavioural operations from other ones in CASL.

▷ CASL does not allow hidden and behavioural equivalence.

The CASL methodology is to provide a notion of behavioural refinement between CASL specifications, based on behavioural equivalence [3].

▷ CASL does not provide built-in transitions.

The carriers in CASL are just sets, rather than (thin) categories, and operations are not regarded as functors.

3.2. Structured Specifications

▷ CASL does not allow translation to derived operations.

In CASL one may concisely define the desired operations in terms of the original ones, and subsequently hide the latter.

▷ CASL does not allow different modes of imports and parameters.

The only mode provided by CASL corresponds to 'protecting' in CafeOBJ. For loose specifications there is not so much difference between the 'protecting', 'extending', and 'using' modes of CafeOBJ. The effect of the latter two modes for extending specifications with initial semantics can generally be obtained delaying the initial semantics until after the extension (although this may require naming extra loose specifications). The methodological implications of adding new values, or identifying old ones, in a specification with initial semantics appear to be not so clear.

▷ CASL does not allow parameters to be implicit.

The rule in CASL is that whenever reference is made to a named specification, fitting arguments have to be provided for all parameters. This should help both readers and writers of parameterized specifications to keep track of just which parameters have to be instantiated. Note that in practice, parameter specifications are often themselves named (or simply the basic specification '**sort** *Elem*') so partial instantiation can still be expressed quite concisely.

▷ CASL does not allow parameters to be labelled.

The symbols declared by the parameters of a generic specification are used directly in the body of the specification, without any distinguishing dot-notation. This is consistent with the CASL treatment of instantiation, where a symbol declared by more than one parameter has to be instantiated uniformly by each argument.

3.3. Convenience Features

▷ CASL does not allow long forms of keywords.

The keywords used in the CASL concrete syntax are mostly quite short, many of them being abbreviations of English words. Keywords at the beginning of lists of declarations, etc., can generally be used in either singular or plural form (without regard to the number of items in the list) but otherwise, each keyword has a unique spelling.

4. The Differences: CASL \ CafeOBJ

The features of CASL that are missing from CafeOBJ include explicit treatment of partiality, general first-order axioms, some convenient abbreviations, and several constructs for structuring specifications and implementations.

Adding some of these features to CafeOBJ (e.g., first-order axioms) would probably hinder execution of specifications using term rewriting; other constructs could probably be added without any significant problems.

4.1. Basic Specifications

▷ CASL allows sort generation constraints.

A sort generation constraint in CASL is essentially a sentence, but it is specified by prefixing a group of signature declarations with the keyword '**generated**'. The effect is not only to eliminate the possibility of 'junk' in the carrier of each declared sort, but also to identify which operations are sufficient basis to generate all the values of the sort. This ensures the soundness of structural induction proofs with cases for the generating operations. In contrast to initiality and free extension, the values of the generated sorts may later be identified without violating the constraint.

▷ CASL allows unrestricted first-order formulae.

CASL provides standard notation for first-order quantifiers and logical connectives, including negation and disjunction. In practice, the majority of specifications will probably use only positive Horn-clauses, universally quantified over all variables; such axioms are also known to be theoretically sufficient for specifying all computable functions. Sometimes, however, the extra expressiveness of the quantifiers and connectives can be used to achieve significantly greater conciseness and comprehensibility.

Some readers might here be wondering whether CASL should be called an *algebraic* specification language at all, since the majority of algebraic specification languages hitherto have been limited, like CafeOBJ, to equational or positive conditional axioms, ensuring close connections with fundamental concepts of universal algebra, and keeping contact with the notion of 'high-school algebra'. The traditional focus on initial-algebra semantics of specifications reinforced the impression that these limitations were somehow essential.

But the term 'algebraic specification' has another widely-held interpretation: simply the specification of (classes of) algebras, regardless of the logic employed and the properties of the specified classes. This is the main sense in which it is meant in 'CASL'.[4]

Universal quantification in CASL is written $\forall V : S \bullet F$. Existential quantification is written using $\exists$. The standard logical connectives are written $F_1 \wedge F_2$, $F_1 \vee F_2$, $F_1 \Rightarrow F_2$ (alternatively F_2 *if* F_1), $F_1 \Leftrightarrow F_2$, and $\neg F$; the atomic formulae *true* and *false* are provided too.

4.2. Structured Specifications

▷ CASL allows declared symbols to be hidden.

The hiding of some of the declared symbols in CASL is written:

$$SP \textbf{ hide } SY_1, \ldots, SY_n$$

When a sort is hidden in CASL, all the operations whose profiles involve that sort get hidden too (ensuring that the resulting signature is well-formed).

One may also specify just those symbols that are not to be hidden, i.e., the symbols to be revealed, using a similar construct.

▷ CASL allows specification of architectural structure.

The essential difference between the *structure of a specification* and a *specification of model structure* is reflected in CASL by providing two kinds of specifications, called respectively *structured* and *architectural*.

Structured specifications allow a large specification to be presented in small, logically-organized parts, with the pragmatic benefits of comprehensibility and reusability. In CASL, the use of these constructs has absolutely no consequences for the structure of models, i.e., of the code that implements the specification. For instance, one may specify integers as an extension of natural numbers, or specify both together in a single basic specification; the models of the complete specification are the same.

It is especially important to bear this in mind in connection with parameterized specifications. The definition of a parameterized specification of lists of arbitrary items, and its instantiation on integers, does *not* imply that the implementation has to provide a parameterized program module for generic lists: all that is required is to provide lists of integers (although the implementor is free to *choose* to use a parameterized module, of course). Sannella, Sokołowski, and Tarlecki [15] provide extensive further discussion of these issues.

In contrast, an architectural specification requires that any model should consist of a collection of separate component units that can be composed in a particular way to give a resulting unit. Each component unit is to be implemented separately, providing a decomposition of the implementation task into separate subtasks with clear interfaces.

[4] The design and institution-independent semantics of structured specifications in CASL have a 'strongly-algebraic flavour' also in the sense of universal algebra.

In CASL, an architectural specification consists of a collection of component unit specifications, together with a description of how the implemented units are to be composed. A model of such a specification consists of a model for each component unit specification, and the described composition. See [4] for further details, motivation, and examples.

CafeOBJ provides structured specifications, but since it does not support architectural specifications, there is no way of expressing whether the various modules are intended to be implemented separately (which might be overly general) or together (where the knowledge of the way that they are combined might be exploited, possibly hindering later reuse of the software in other contexts).

4.3. Convenience Features

In CASL, the main purpose of providing abbreviations is conciseness and perspicuity, which are perhaps even more important qualities for the readers of a specification than for its writer(s). A secondary purpose is to provide syntactic support for particular methodologies.

▷ CASL allows *definitions* of subsorts, operations, and predicates.

A subsort definition is written:

$$\textbf{sort}\ S = \{\ V : S' \bullet F\ \}$$

and declares S to be the subsort of S' consisting of just those values of the variable V for which the formula F holds.

A total function definition is written:

$$\textbf{op}\ f(V_1 : S_1; \ldots; V_n : S_n) : S\ =\ T$$

and a constant definition as:

$$\textbf{op}\ c : S\ =\ T$$

CASL also provides similar constructs for defining partial functions and predicates.

▷ CASL allows declarations of datatypes with constructors and selectors.

In a practical specification language, it is important to be able to avoid tedious, repetitive patterns of specification, as these are likely to be carelessly written, and never read closely. The CASL construct of a datatype declaration collects together several such cases into a single abbreviatory construct, which in some respects corresponds to a type definition in STANDARD ML, or to a context-free grammar production in BNF.

A datatype declaration is written

$$\textbf{type}\ S\ ::=\ A_1 \mid \ \ldots\ \mid A_n$$

It declares the sort S, and lists the alternatives A_i for that sort. An alternative may be a constant c, whose declaration is implicit; or it may be a list of sorts, written *sorts* $S_1, \ldots, S_n$, to be embedded as subsorts; or it may be a 'construct' (essentially an indexed product) written $f(\ldots; f_i{:}S_i; \ldots)$, given by a constructor function f together with its argument sorts S_i, each optionally accompanied by a selector f_i. The declarations of the constructors and selectors, and the assertion of the expected axioms that relate them to each other, are implicit.

Datatype declarations may be prefixed by '**generated**' or '**free**'. Both of these provide the appropriate sort generation constraint, and the latter construct ensures moreover that the ranges of the constructors are disjoint. In particular, the free datatype declaration of constants:

free type $S ::= c_1 \mid \ldots \mid c_n$

corresponds to an ordinary enumerated type as found in programming languages (although no implicit successor function is provided here).

▷ CASL allows conditional terms.

A conditional term is written 'T_1 *when* F *else* T_2', where the condition is a formula F. Its use in a term abbreviates a formula of the form:

$$(F \Rightarrow \ldots T_1 \ldots) \wedge (\neg F \Rightarrow \ldots T_2 \ldots)$$

It appears that use of such a construct would provide a significant abbreviation in specifications following the style of some of those illustrated in the CafeOBJ Report [10, pp. 153–155].

▷ CASL allows omission of repeated keywords.

An item of a specification is assumed to be of the same kind as the previous item, unless an explicit keyword indicates otherwise. For instance, it is sufficient to write the keyword '**ops**' just once at the beginning of a list of separate operation declarations.

▷ CASL allows display annotations.

CASL provides a uniform way of influencing the way that (user-declared) symbols are to be displayed. CASL input syntax for symbols is restricted to ISO Latin-1 characters, but display annotations allow mathematical and other symbols to appear in the output, potentially enhancing readability.

▷ CASL allows local specifications.

A local specification allows declared symbols to be automatically hidden after use. It is written:

local SP_1 **within** SP_2

▷ CASL allows compound identifiers.

CASL allows the use of compound sort identifiers of the form '$I[\ldots, I_i, \ldots]$' in generic specifications. For instance, instead of using just *List* for the sort of lists, it may be a symbol formed with the sort of items *Elem* as a component: *List*[*Elem*]. The fitting of the parameter sort *Elem* to an argument sort affects this compound sort symbol for lists too, giving distinct symbols such as *List*[*Int*], *List*[*Char*] when instantiating lists with integers *Int* and characters *Char*, and thereby avoiding the danger of unintended identifications (and the need for explicit renaming) when combining such instantiations.

Note that CafeOBJ allows instead the disambiguation of sort names by qualifying them with module expressions that identify their origins.

▷ CASL allows libraries of specifications to be named and referenced.

In CASL, libraries of definitions may be named and installed on the Internet. A library may specify the downloading of named definitions from named libraries, possibly providing a new local name. Downloading from previous versions of libraries can be indicated. Local libraries, which have not yet been installed for global access, are referred to temporarily by their URLs.

The CafeOBJ system also supports libraries, but this feature is external to the CafeOBJ language (and thus not described in the CafeOBJ report).

5. Conclusion

Taking into account the different aims and methodologies espoused by the designers of CafeOBJ and CASL, the correspondence between the equational fragments of the two languages is perhaps as close as could reasonably be expected—especially in view of the intended close relationship between CafeOBJ and OBJ3, and the COFI requirement for CASL to be unbiased towards any particular previous framework. Some of the differences could easily be eliminated, e.g., datatype declarations might be added to CafeOBJ, derived operations might be added to CASL. Others would require more work, e.g., adding syntactic and semantic support for rewriting logic to CASL. It appears that some problems with the representation of predicates in CafeOBJ might be removed by adopting an approach closer to that adopted for CASL; similarly for overloading, although the mixing of grouping analysis and overloading resolution in CafeOBJ may make the implementation of CASL-style overloading less tractable.

Acknowledgements

Christine Choppy, Răzvan Diaconescu, Kokichi Futatsugi, and Till Mossakowski provided useful comments on an earlier version of this paper, as did several participants of the CafeOBJ Symposium at which it was presented. The five anonymous referees pointed out several inaccuracies in the submitted version, and made many useful suggestions for improvement. Special thanks to the participants of the various COFI task groups for the collaborative effort which has resulted in the design of CASL. The author gratefully acknowledges the support of BRICS (Basic Research in Computer Science, a centre of the Danish National Research Foundation).

REFERENCES

1. C. Alexander. *A Timeless Way of Building.* Oxford University Press, 1979.
2. E. Astesiano, H.-J. Kreowski, and B. Krieg-Brückner, editors. *Algebraic Foundation of Systems Specification.* IFIP State-of-the-Art Reports. Springer-Verlag, 1999.
3. M. Bidoit, D. Sannella, and A. Tarlecki. Behavioural encapsulation. Language Design Study Note MB+DTS+AT-1, in [5], 1996.
4. M. Bidoit, D. Sannella, and A. Tarlecki. Architectural specifications in CASL. In *AMAST '98, Proc. 7th Intl. Conference on Algebraic Methodology and Software Technology, Manaus*, volume 1548 of *LNCS*, pages 341–357. Springer-Verlag, 1998.
5. CoFI. The Common Framework Initiative for algebraic specification and development, electronic archives. Notes and Documents accessible from `http://www.brics.dk/Projects/CoFI`.
6. CoFI Language Design Task Group. Response to the Referee Report on CASL. Documents/CASL/RefereeResponse, in [5], Aug. 1997.
7. CoFI Language Design Task Group. CASL – The CoFI Algebraic Specification Language – Design Proposal. Documents/CASL/Proposal, in [5], May 1997.
8. CoFI Language Design Task Group. CASL – The CoFI Algebraic Specification Language – Summary. Documents/CASL/Summary, in [5], Oct. 1998.
9. CoFI Semantics Task Group. CASL – The CoFI Algebraic Specification Language – Semantics. Note S-9 (version 0.95), in [5], Mar. 1999.
10. R. Diaconescu and K. Futatsugi. *CafeOBJ Report: The Language, Proof Techniques, and Methodologies for Object-Oriented Algebraic Specification*, volume 6 of *AMAST Series in Computing.* World Scientific, 1998.
11. A. Haxthausen, B. Krieg-Brückner, and T. Mossakowski. Extending CASL with higher-order functions — design proposal. Note L-8 (see also Note L-10), in [5], Jan. 1998.
12. H. Kirchner and C. Ringeissen. Executing CASL equational specifications with the ELAN rewrite engine. Note T-9, in [5], Oct. 1999.
13. T. Mossakowski. Sublanguages of CASL. Note L-7, in [5], Dec. 1997.
14. T. Mossakowski. Two "functional programming" sublanguages of CASL. Note L-9, in [5], Mar. 1998.
15. D. Sannella, S. Sokołowski, and A. Tarlecki. Toward formal development of programs from algebraic specifications: Parameterisation revisited. *Acta Inf.*, 29:689–736, 1992.

A. Appendix: CASL Overview and Examples

This section gives a concise overview of all the main CASL features, covering both that are in common with CafeOBJ as well as those that are not.

▷ Basic specifications in CASL list declarations, definitions, and axioms.

Functions are partial or total, and predicates are allowed. Subsorts are interpreted as embeddings. Axioms are first-order formulae built from definedness assertions and both ordinary and existential equations. Sort generation constraints can be applied to groups of

declarations. Datatype declarations are provided for concise specification of enumerations, unions, and products.

▷ Structured specifications in CASL allow translation, reduction, union, and extension of specifications.

Extensions may be required to be conservative and/or free; initiality constraints are a special case. A simple form of generic specification is provided, together with instantiation involving parameter-fitting translations that affect compound identifiers.

▷ Architectural specifications in CASL express implementation structure.

The specified software is to be composed from separately-developed, reusable units with clear interfaces.

▷ Libraries in CASL provide collections of named specifications.

Downloading involves retrieval of specifications from distributed libraries.

CafeOBJ Examples in CASL

The following specifications illustrate how one might translate some of the examples given in the CafeOBJ Report [10] into CASL. See also the examples given at the beginning of Sect. 2 (Example 12 in [10]) for an illustration of the declaration of partial operations.

Example 1

```
mod! BARE-NAT {
  [ NzNat Zero < Nat ]

  op 0 : -> Zero
  op s_ : Nat -> NzNat
}
```

CASL:

spec BARENAT =
free types
$Nat ::= 0 \mid sort\ NzNat;$
$NzNat ::= s\_\_(Nat)$

Example 23

```
mod! SIMPLE-NAT {
  protecting(BARE-NAT)

  op _+_ : Nat Nat -> Nat {comm}

  eq s(N:Nat) + M:Nat = s(N+M) .
  eq N:Nat + 0 = N .
}
```

CASL:

spec SIMPLENAT =
  BARENAT **then**
  op $\_\_ + \_\_ : Nat \times Nat \to Nat, comm$
  vars $M, N : Nat$
  • $s(N) + M = s(N + M)$
  • $N + 0 = N$

Example 26

```
mod* MON* (X :: TRIV) {

  op nil : -> Elt
  op _;_ : Elt Elt -> Elt {assoc idr: nil}
}

mod* CMON* (Y :: MON) {

  op _;_ : Elt Elt -> Elt {comm}
}
```

CASL:

spec MON [**sort** *Elt*] =
  ops $nil : Elt$;
    $\_\_ * \_\_ : Elt \times Elt \to Elt, assoc, unit\ nil$

spec CMON [MON [**sort** *Elt*]] =
  op $\_\_ * \_\_ : Elt \times Elt \to Elt, comm$

Example 27

```
mod* MON-POW (POWER :: MON, M :: MON)
{
  op _^_ : Elt.M Elt.POWER -> Elt.M

  vars m m' : Elt.M
  vars p p' : Elt.POWER

  eq (m ; m')^ p  = (m ^ p) ; (m' ^ p) .
  eq m ^ (p ; p') = (m ^ p) ; (m ^ p') .
  eq m ^ nil      = nil .
}
```

CASL:

spec MONPOW [MON [**sort** *Elt*]] [MON [**sort** *Pow*]] =
op $\_\_ \uparrow \_\_ : Elt \times Pow \to Elt$
vars $e, e' : Elt;\ p, p' : Pow$
- $(e * e') \uparrow p = (e \uparrow p) * (e' \uparrow p)$
- $e \uparrow (p * p') = (e \uparrow p) * (e \uparrow p')$
- $e \uparrow nil = nil$

Chapter 7

CafePie: A Visual Programming System for CafeOBJ

Tohru Ogawa * and Jiro Tanaka [a†]

[a]Institute of Information Sciences and Electronics, University of Tsukuba, Tsukuba, Ibaraki 305-8573, Japan

CafePie is a visual programming system for CafeOBJ, an algebraic specification language based on term rewriting. Program editing and execution in CafePie are performed in one window. All program editing operations are handled in a uniform manner.

An abstract visualization schema is necessary to understand the program at the programming language level. In this paper, we propose visualized term rewriting with more realistic expressions. With our approach, users can customize the term expression as they like by using visual transformation rules. These rules can also be edited using drag-and-drop operations.

1. Introduction

"CafePie" [1–4], or CafeOBJ Pictorial Interactive Editor, is a visual programming system (VPS [5]). CafePie is a visual interface for the term rewriting portion of CafeOBJ. CafePie can be used as an environment for program editing. It visualizes each module of the CafeOBJ program and allows the user to edit it visually. Combined with the CafeOBJ interpreter, it can also serve as a visual rewriting environment for CafeOBJ.

Direct manipulation [6] is performed by using a mouse with CafePie. Each visualized element can be moved in accordance with the mouse movement. The same visualization schema is used for both program editing and execution. Since the program editing and execution are performed in one window, it is possible to directly reflect any program modification onto the program execution.

2. Features of Visual Programming

A visual language manipulates visual information or supports visual interaction, or allows programming with visual expressions. The latter is taken to be the definition of a visual programming language (VPL). VPLs may be further classified according to the type and extent of visual expression used, into icon-based languages, form-based languages and diagram languages [7]. Naturally visual languages have an visual expression for which there is no obvious textual equivalent [8]. Two-dimensional displays for programs, such as flowcharts and a program visualization system Incense [9], have long been known to

*tohru@softlab.is.tsukuba.ac.jp
†jiro@is.tsukuba.ac.jp

be helpful aids in program understanding. In addition, there have been much research on executable VPS, such as Pict [10], HI-VISUAL [11] and PP [12].

In general, a more visual style of programming could be easier to understand and generate for humans, especially for non-programmers or novice programmers since visual programming (VP) can be presented attractively. Moreover, the style is useful in software specification area, such as component based software [13,14]. A tool qualifies as a visual programming if it is possible to build some application without textual programming. However, currently very few practitioners use VP.

CafePie is based on the algebraic specification language (ASL) CafeOBJ, which is a high-level declarative programming language. A declarative programming language is suitable for visualization by a VPS because visual programming is also declarative. Visual expressions using the box-and-line representation have less commitment to the order of interpreting code than textual expressions. One can regard visual expressions to be declared spatially. A specification language requires comparatively fewer programming elements than a procedural programming language, and therefore specification language can be visualized with fewer kind of icons. Generally speaking, an environment with a user-friendly graphical interface has the advantage of enabling easy interpretation of the structures of the terms and rewriting processes.

3. The "CafePie" System

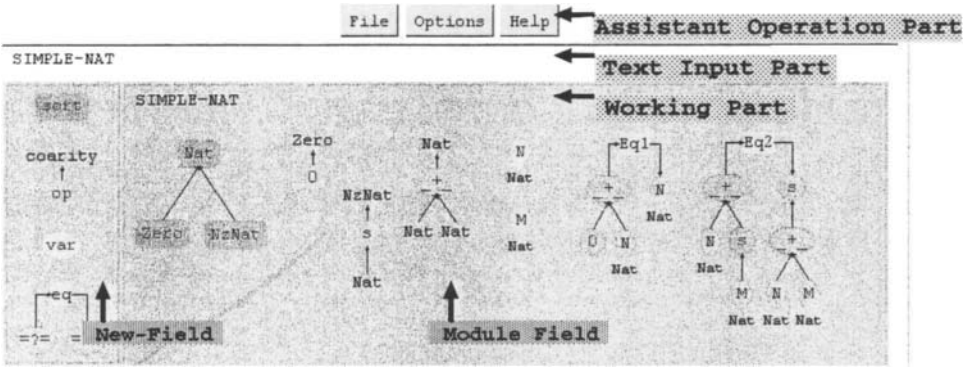


Figure 7.1. A Snapshot of CafePie

We developed CafePie and implemented it in Java. CafePie was developed in Java Development Kit (JDK™) version 1.0 at the first implementation, but the version is now 1.2. CafePie is ordinarily implemented as a Java application. In this version, users can edit and execute programs in the system. CafePie is also implemented as an applet on a Web browser. The applet version is used only for program editing.

Figure 7.1 shows a snapshot of CafePie. The upper part of the figure consists of buttons and is the "Assistant Operation Part." This enables the user to load or save a file (File button), set the CafeOBJ server (Options button), and view textual guides (Help button).

The "Text Input Part" is used to input the label of each editable icon. The main part of the figure shows the programming space called the "Working Part." The user edits a CafeOBJ program in this part. The "Module Field" in the Working Part shows the current CafeOBJ module to edit. The left side of the Module Field is the "New-Field," which consists of essential icons such as sort, operator, variable, and equation. This field is used to make a new icon in the Module Field.

```
module SIMPLE-NAT {
 [ Zero NzNat < Nat ]
  signature {
    op 0 : -> Zero
    op s : Nat -> NzNat
    op _+_ : Nat Nat -> Nat { comm, assoc }
  }
  axioms {
    var N : Nat
    var M : Nat
    eq [0] : 0 + N = N .
    eq [1] : N + s(M) = s(N + M) .
  }
}
```

Figure 7.2. "simple-nat.mod" -CafeOBJ Program File-

The following functions have been implemented in CafePie:

- Input program objects by figures.
  Users can input each basic object of an ASL language using an icon. These icons can be edited by direct manipulation.

- Generate visual icons from the codes automatically.
  Users can input a textual expression, and the system will generate icons from the expression.

- Editing visual objects.
  Visual expressions can be edited at any time. Users can program visually using this function while they edit or revise programs that have already been generated from the textual programs of CafeOBJ.

- Program save/load.
  Visually-edited programs can be saved to a file. Users load the file when necessary. CafePie saves the visual expressions after it converts them to CafeOBJ program expressions.

- Program execution.
  A goal (-term) represented by the visual icons can be executed. In this case, CafePie is connected to the CafeOBJ interpreter. CafePie behaves like a visual interface in the program execution.

For example, the file "simple-nat.mod" (Fig.7.2), which is a specification of natural numbers (under addition) written in CafeOBJ, is loaded by clicking on the File button. The program, visualized with pictorial objects, then appears in the Module Field of the Working Part (as in Fig. 7.1). The visualized program can be edited by direct manipulation. If the edited program is saved, a CafeOBJ program file and another textual file that contains layout information are created.

3.1. Program Visualization in CafePie

"Visualization of program structure" means expressing the program structure using pictorial or graphical objects. We visualize the program structures of CafeOBJ by expressing the program elements with pictorial objects. Each pictorial object is called an "Icon." We have chosen the following primitive elements for CafeOBJ: sort, operator, variable, and equation.

The visualization rules for each element are presented below.

Sort CafePie uses a directed graph to depict the sort orders. The sorts are represented in Fig.7.3 by green rectangular nodes (only shaded rectangles are seen in the manuscript) and the orders are represented by directed edges (Table 7.1).

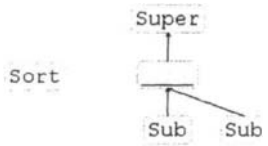


Figure 7.3. Sort Icon and Sort Relation

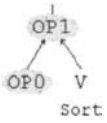
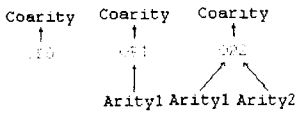


Figure 7.4. Term Icon

Term A term is formed with operators and variables. The structure of a term is displayed as a tree. Figure 4 shows the tree structure of the term "OP1 (OP0, V:Sort)." A component of a term, i.e. an operator or a variable, is represented by a node, and an arrow is drawn from the term to its superterm to express the super-sub relation between these components.

Operator and variable An operator is denoted by an operator symbol, its sort "coarity," and its attributes "arities." An operator is represented in Fig.7.5 by a light blue oval and

has a label for the operator symbol (Table 7.1). The labels of the arities are arranged at the bottom part of the operator, and the label of the coarity is arranged at the top part of the operator. Arrows are drawn from arities to operator and from operator to coarity. A variable, which appears in Fig.7.6, is represented by an orange oval (Table 7.1), and the sort of the variable is represented at its lower part.



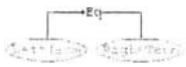


Figure 7.5. Operator Icon Figure 7.6. Variable Icon Figure 7.7. Equation Icon

Equation CafePie is mainly concerned with the operational semantics of CafeOBJ, so equations are always regarded as rewrite rules. A label is arranged in the center top of the equation, as shown in Fig.7.7. The left side is arranged on the bottom left side of the label, and the right side is on the bottom right. Arrows from the left term to the label and from the label to the right term are drawn to form a balanced shape to represent a term rewriting rule (Table 7.1).

Table 7.1
Icons' Colors and Shapes in CafePie

Icon	Sort	Term	Operator	Variable	Equation	Module
Color	Green	-	Light blue	Orange	White (label)	Gray
Shape	Rectangle	(Tree)	Oval	Oval	(Balance)	Field (Rectangle)

Module Field A CafeOBJ program consists of modules. A module is represented as a gray rectangle called a "field" (Table 7.1). The module contains other primitive elements: sort, operator, variable, and equation. We can edit these primitive elements.

3.2. Drag-and-Drop-based Program Editing

We use direct manipulation to implement program editing. Direct manipulation is easy to learn, and the user can immediately recognize any mistakes. Complex and obscure operations can cause unexpected consequences; simple operations enable more smooth program editing.

All icon-editing operations are handled in a uniform manner, using a drag-and-drop operation [15]. This drag-and-drop technique is well known for its simplicity. For icon movement, the user moves the icon using the drag-and-drop technique. If an icon already exists where the user wants to drop the icon, the two icons will overlap. Overlapping two

icons with the drag-and-drop technique is important in the editing process. The process of the drag-and-drop method consists of

1. Selecting an icon,
2. Moving (or dragging) the selected icon to another icon, and
3. Overlapping (or dropping) the selected icon with another icon.

The target icon moves with the mouse cursor and remains visible throughout the movement. The user moves the icon by dragging it, without losing sight of what he is doing. We reexamined this technique to realize program editing. Program editing operations in CafePie involve making/deleting a relation between two sorts, adding/changing an arity of an operator, and creating/adding a subterm on a variable. Table 7.2 shows these program editing operations. An event is invoked when an icon (*source*) is overlapped onto another icon (*target*). After the event is invoked, the action corresponding to the event is carried out. The program editing process is the repetition of these elementary actions.

Table 7.2
Drag-and-Drop-based Program Operations in CafePie

Event Name	Source	Target	Action
Make Sort-Relation	Sort	Sort	Relate one sort to another (as supersort)
Delete Sort-Relation	Sort	Sort	Delete the relation between two sorts
Add Arity	Sort	Operator	Add an arity to an operator
Change Arity	Sort	Arity	Change the arity to one that has the sort name
Change Coarity	Operator	Sort	Change the coarity to one that has the sort name
Exchange Arities	Arity	Arity	Exchange one arity for the other
Create Subterm	Operator	Variable	Replace the variable with a new term
Add Subterm	Term	Variable	Replace the variable with the (copied) term

For example, operator "`s`," which appears in the sample code SIMPLE-NAT, has an arity sort called "`Nat`" ("`op s : Nat -> NzNat`"). This operator is created from several steps.

- First, an operator icon that has no arity (constant) is created by default.
- Next, the sort icon "`Nat`" which has already been defined is moved toward the operator.
- Finally, these two icons are overlapped, the "Add Arity" event (in Table 7.2) is carried out, and the arity sort called "`Nat`" is added to the operator.

Another example is called "Create Subterm." The left term of the equation "`1`," which appears in the SIMPLE-NAT, is "`N:Nat + s(M:Nat)`." Operators "`_+_`" and "`s`" are used to create this term.

- Suppose there is a variable that belongs to the sort "`Nat`."

- Moving the operator "_+_" onto the variable changes the variable to the term "`V1:Nat + V2:Nat`".
- Similarly, moving the operator "`s`" onto the variable "`V2`" (of the term) changes the variable "`V2`" to the term "`V1:Nat + s(V2:Nat)`".

In this way, the drag-and-drop technique is applied to CafePie. All operations of the program editing are handled in a uniform manner.

3.3. Program Execution in CafePie

CafePie enables the program execution by combining with CafeOBJ interpreter. In order to utilize the interpreter, CafePie must communicate with "cafemaster," which is a network server for CafeOBJ. CafePie and the interpreter are connected by cafemaster. (Cafemaster has two modes for combining a client with the interpreter, i.e, the session mode and the interactive mode. In the current implementation, CafePie accesses the interpreter in the interactive mode.)

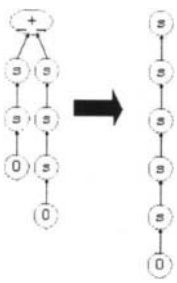
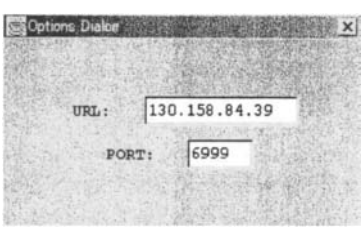


Figure 7.8. A Goal Term for Program Execution



`URL: Users can designate an IP address or URL as the location of the interpreter.`

`PORT: Users can input the port number of the CafeOBJ interpreter.`

Figure 7.9. Options Dialog of CafePie

- **Edit a goal term:**
  A user edits a term (goal) in the Module Field. It is called a goal and is used to test the module SIMPLE-NAT. For example, we create the goal "`s(s(0)) + s(s(s(0)))`" (the left side of Fig.7.8).

- **Start the term rewriting:**
  A program consists of a module displayed in the Module Field. Each module has a label. The label is drawn at the upper left of Module Field (Fig.7.1). The user invokes evaluation (program execution) by moving the term onto the label.

- **Connect to the interpreter:**
  CafePie tries to connect to the interpreter running on a remote host by using socket communication. If a connection is achieved, CafePie connects to the interpreter in an interactive mode (CafePie sends a message "`interactive`" to the interpreter). Users can specify the interpreter's network address. They click on the Options button of the Assistant Operation Part, and an options dialog appears on CafePie (Fig.7.9). The IP address and the port of the CafeOBJ interpreter are designated in the dialog. Thereafter, CafePie knows where the interpreter is.

- **Send the module information to the interpreter:**
  After connecting to the interpreter, CafePie converts the module's visual expression into a text-based CafeOBJ program and sends the program to the interpreter. The information is comprised of a module name, sorts, operations, variables and equations (Fig.7.2).

- **Send the goal to the interpreter:**
  After sending the program, CafePie sends the goal term "`s(s(0)) + s(s(s(0)))`" to the interpreter. CafePie orders the interpreter to start the program execution (CafePie sends two messages, "`set trace on`" and "`red s(s(0)) + s(s(s(0)))
  .`", Fig.7.12).

- **Receive the result from the interpreter:**
  The goal is rewritten repeatedly on the interpreter. CafePie receives the term rewriting trace as a result after execution is completed (Fig. 7.12). The tracing result consists of terms that illustrate the process of reductions. The result is processed by CafePie and is shown in the visualized form.

CafePie shows the terms in succession like an animated cartoon. This is a dynamic representation and is suitable for checking the rewriting flow at any time. Figure 7.10 shows the process of term rewriting when the goal term is "`s(s(0)) + s(s(s(0)))`" of the module SIMPLE-NAT and the rewritten term is "`s(s(s(s(s(0)))))`" (the right side of Fig. 7.8). This is an effective dynamic representation of the term rewriting process. After showing the last term, CafePie presents the tracing diagram in the shape of an obi (an obi is a Japanese broad sash tied over a kimono, Fig. 7.11). This is a static display and is suitable for checking one reduction process more closely.

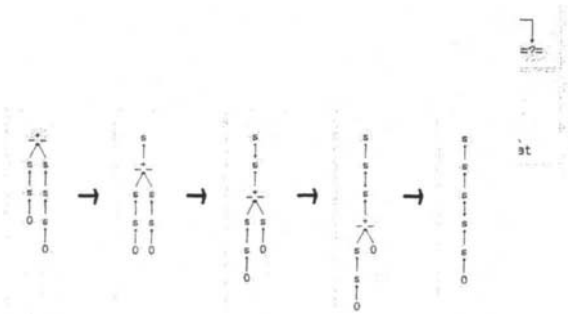


Figure 7.10. Dynamic Representation

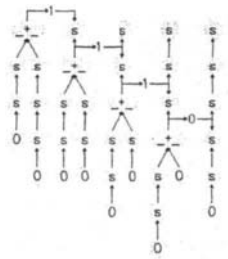


Figure 7.11. Static Representation

```
SIMPLE-NAT> set trace on

SIMPLE-NAT> red s(s(0)) + s(s(s(0))) .
 -- reduce in SIMPLE-NAT : s(s(0)) + s(s(s(0)))
 1>[1] rule: eq N:Nat + s(M:Nat) = s(N:Nat + M:Nat)
      { N:Nat |-> s(s(0)), M:Nat |-> s(s(0)) }
 1<[1] s(s(0)) + s(s(s(0))) --> s(s(s(0)) + s(s(0)))
 1>[2] rule: eq N:Nat + s(M:Nat) = s(N:Nat + M:Nat)
      { N:Nat |-> s(s(0)), M:Nat |-> s(0) }
 1<[2] s(s(0)) + s(s(0)) --> s(s(s(0)) + s(0))
 1>[3] rule: eq N:Nat + s(M:Nat) = s(N:Nat + M:Nat)
      { N:Nat |-> s(s(0)), M:Nat |-> 0 }
 1<[3] s(s(0)) + s(0) --> s(s(s(0)) + 0)
 1>[4] rule: eq 0 + N:Nat = N:Nat
      { N:Nat |-> s(s(0)) }
 1<[4] s(s(0)) + 0 --> s(s(0))
 s(s(s(s(0)))) : NzNat
 (0.010 sec for parse, 4 rewrites(0.070 sec), 10 match attempts)

 SIMPLE-NAT>
```

Figure 7.12. An Execution Result of the CafeOBJ Interpreter

3.4. Realistic Visualization

The process of term rewriting is visualized as tree structures that consist of icons. For example, Fig.7.13 shows the result of visualization of the term "`push( E3:Elt, push(`

E2:Elt, push(E1:Elt, push(E0:Elt, empty))))," by CafePie. The specification that begets this term is expressed as module STACK (Fig.7.15).

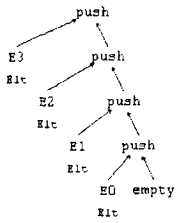


Figure 7.13. Original Stack Visualization

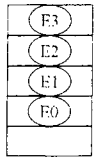


Figure 7.14. New Stack Visualization

```
module STACK [X::TRIV] {
 [ NeStack < Stack ]
  signature {
    op empty : -> Stack
    op push : Elt Stack -> NeStack
    op pop : NeStack -> Stack
    op top : NeStack -> Elt
  }
  axioms {
    var S : Stack
    var E : Elt
    eq pop( push( E, S ) ) = S .
    eq top( push( E, S ) ) = E .
  }
}
```

Figure 7.15. "stack.mod" –CafeOBJ Program File–

This visualization method is difficult for users to understand in an intuitive manner because users in general mentally visualize stacks not as trees but as building blocks. More realistic expressions of higher abstraction levels are desired.

Visual Transformation Rule and Term Visualization Example (1)

We propose a method that visualizes term rewriting with more realistic expressions, by using figures, pictures, and images. We call these expressions visual objects. The visual objects should be edited without the program code showing. In the real world, the essential characteristic of an actual object is its shape. We propose to use visual transformation rules so that users can change the shapes of visual objects.

Rewriting rules are called "equations" in CafeOBJ. An equation is composed of operators and variables. CafePie can change the term representation. Specifically, it can change the system-prepared view to a user-defined view by using visual transformation rules. For example, the STACK program of CafeOBJ has the operators "`empty`" and "`push`." By default, the expression of these operators have been prepared by the system (the left part of Fig.7.16 and 7.17). If a user imagines that the STACK is like building blocks, the result of STACK visualization would be like building blocks. The operator "`empty`" is represented by a rectangle (the right part of Fig. 7.16) to imitate building blocks, instead of the original visualization (the left part of Fig. 7.16). The operator "`push`" is visualized like the right part of Fig. 7.17. This figure shows that the rectangle with "Elt" is arranged at the upper part of "Stack." After defining these rules, the old terms (Fig. 7.13) are changed to other expressions, as in Fig. 7.14.

Figure 7.16. Empty Operator Visualization(1)

Figure 7.17. Push Operator Visualization(1)

Defining the Visual Transformation Rule

We have developed an environment in which users can edit the visual transformation rules in CafePie by using direct manipulation. In our approach, these rules are not defined by using a drawing action but by using a combination of prepared visual objects. Therefore, the users can easily define the rules and edit programs in the same paradigm. Editing the rules requires the two steps below.

1. Preparing the visual objects. The system has some elementary figures such as rectangles and circles, and images that are loaded from files. If an operator has arities, users can also use them as visual objects.

2. Defining the geometrical relations. The user creates a repeated relation between two objects to define the geometrical relations. The users can also treat related objects as one object.

The relation is provided by the drag-and-drop operation. We use a "plate"-node icon. When a node is dropped onto the plate, the node is added on the plate, and its position is arranged automatically. Suppose the user moves a node toward the plate-node. When the user drops the node onto the plate, dotted lines appear around the plate, as shown in Fig.7.18. These lines indicate the expected location of the node. The node's location is selected by default from any of the nine parts of the plate: the upper left, the upper middle, the upper right, the left side, the center, the right side, the lower left, the lower middle, or the lower right. The location of the dropped node is determined as follows:

- If the node is dropped on the upper or lower part of the plate, the node is designed to stick to the plate.
- If the node is dropped on the right or left side of the plate, they are also arranged to be close together.
- If the node is dropped in the diagonal part of the plate, the node center is arranged on the vertex of the plate.

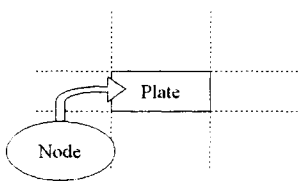


Figure 7.18. Make Visual Transformation Rule

The size of the dropped node is determined by the node's location.

- If the user drops the node in the center of the plate, the node's size becomes smaller than the plate. The plate contains the node.
- If the node is dropped in the left or right part of the plate, the node's height is modified to have the same height as the plate.
- If the node is dropped in the upper or lower part of the plate, the node's width is modified to have the same width as the plate.
- If the node is dropped in the diagonal part of the plate, the node's size is changed to be the same size of the plate.

Term Visualization Example (2)

Another visualization method can be applied to STACK instead of the example using building blocks.

The visual transformation rules of the operators "`empty`" and "`push`" can be re-defined. The right hand side of Fig.7.19 shows the new rule of the operator "`empty`." This figure indicates "No Exit" because the Exit door has broken down. The right hand side of Fig.7.20 shows the new rule of the operator "`push`." This figure indicates that a person who has a face "`Elt`" is in the rear of the "`Stack`." Figure 7.21 shows a term according

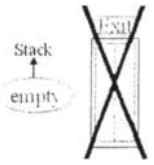


Figure 7.19. Empty Operator Visualization(2)

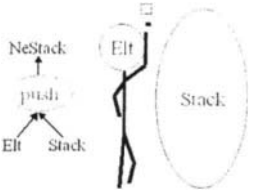


Figure 7.20. Push Operator Visualization(2)

to the new visualization rules. Each person has a different expression. No person can go forward because of the broken door. Only the person who is at the end of the line can move. This mechanism represents the STACK structure. In this visualization, STACK represents a line of people. Programs can be expressed differently in this way by defining different visual transformation rules.

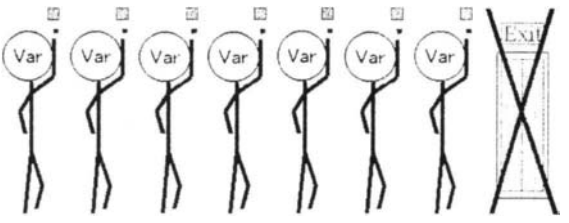


Figure 7.21. New Stack Visualization (2)

4. Related Works

Various systems have been proposed through which users can watch and analyze the term-rewriting system (TRS). ReDux [16] is a workbench for TRS realized by a textual interface. ReDux has various interfaces with completion algorithms. They came up with various concepts in the text interface. However, users cannot manipulate the terms intuitively. TERSE [17] is a visual support environment for TRS. The system can visually show the process of term rewriting. The system supports the environment for program execution, but does not support program editing. CafePie visually supports not only program execution but also program editing. Users often understand the program through the execution and want to subsequently re-edit the program. Our main point is that CafePie can edit and execute the program visually. CafePie is the first system that shows TRS execution dynamically. Viry presents some preliminary ideas towards a user interface for completion and its integration within programming environments [18].

SDL [19], G-LOTOS [20,21] and Petri Nets [22] are graphics-based specification languages. SDL is a specification language with both graphical and character-based syntaxes for defining interacting extended finite state machines, and is used to specify discrete interactive systems such as industrial process control, traffic control, and telecommunication systems. G-LOTOS, which has two-dimensional constructions, enables LOTOS to express the specification diagrammatically. Petri Nets is applied to the modeling and analysis of computer architecture problems, and has a graphical and formal syntaxes.

In addition, various kinds of VPLs have been proposed. Form/3 [23] is a declarative, form-based, language that follows the spreadsheet paradigm. ChemTrains [24] is a rule-based language in which both the condition and action of each rule are specified by pictures.

5. Summary and Further Research

We have implemented CafePie, a VPS for CafeOBJ. The module structures are visualized with icons and can be edited intuitively using the drag-and-drop technique. The execution process of the program, which is the term-rewriting process for the initial term, is also visualized with icons. Program execution is described by using the same iconic descriptions as in program editing. Term rewriting is visualized with realistic expressions by using figures, pictures, and images. We map operators to realistic expressions so that equations are expressed as transformations of realistic expressions. We use visual transformation rules that give the program pictorial expressions so that users can customize the term expression to their preference.

Our system, CafePie, is useful for ASL beginners. Our goal is to improve the system and to fascinate advanced users. Shneiderman [25] stated that direct manipulation is not appropriate when the data structure to be displayed is large, which can very easily happen with an application to algebraic specification. Another issue that must be discussed is the help for browsing specifications, which in CafeOBJ has a modular structure, because research has shown that users spend a lot of time trying to get their specifications just right.

REFERENCES

1. T. Ogawa and J. Tanaka. Drag and Drop based Visual Programming Environment for Algebraic Specification Language. In *15th Conference Proceedings Japan Society for Software Science and Technology(JSSST-98)*, pages 165–168, 1998. (in Japanese).
2. T. Ogawa and J. Tanaka. Double-Click and Drag-and-Drop in Visual Programming Environment for CafeOBJ. In *Proceedings of International Symposium on Future Software Technology (ISFST'98)*, pages 155–160, Hangzhou, October 28-30 1998.
3. T. Ogawa and J. Tanaka. Realistic Program Visualization in CafePie. In *Proceedings of World Conference on Integrated Design and Process Technology (IDPT'99)*, 1999. (to appear).
4. T. Ogawa and J. Tanaka. CafePie: CafeOBJ Visualization by using a Combination of Diagrams. In *16th Conference Proceedings Japan Society for Software Science and Technology(JSSST-99)*, pages 65–68, 1999. (in Japanese).
5. B. A. Myers. Taxonomies of Visual Programming and Program Visualization. *Journal of Visual Languages and Computing*, 1(1):97–123, 1990.
6. B. Shneiderman. Direct Manipulation: A Step Beyond Programming Languages. *IEEE Computer*, 16(8):57–69. 1983.
7. E. J. Golin and S. P. Reiss. The Specification of Visual Language Syntax. *Journal of Visual Languages and Computing*, 1(2):141–157, 1990.
8. A. L. Ambler and M. M. Burnett. Influence of Visual Technology on the Evolution of Language Environments. *IEEE Computer*, 6(2):9–22, October 1989.
9. Brad A. Myers. Incense: A System for Displaying Data Structures. *Computer Graphics: SIGGRAPH '83 Conference Proceedings*, 17(3):115–125, July 1983.
10. E. Glinert and S. Tanimoto. PICT: An Interactive Graphical Programming Environment. *IEEE Computer*, 17(11):7–25, 1984.
11. M. Hirakawa, M. Tanaka, and T. Ichikawa. An Iconic Programming System, HI-VISUAL. *IEEE Transaction on Software Engineering*, 16(10):1178–1184, 1990.
12. J. Tanaka. PP : Visual Programming System For Parallel Logic Programming Language GHC. *Parallel and Distributied Computing and Networks '97*, pages 188–193, August 11-13 1997. Singapore.
13. M. P. Stovsky and B. W. Weide. Building Interprocess Communication Models Using Stile. In E. P. Glinert, editor, *Visual Programming Environments: Paradigms and Systems*, pages 566–574. IEEE Computer Society Press, Los Alamitos, 1990.
14. D. C. Smith and J. Susser. A Component Architecture for Personal Computer Software. In B. A. Myers, editor, *Languages for Developing User Interfaces*, pages 31–56. Jones and Bartlett Publishers, Boston, 1992.
15. A. Wagner, P.Curran, and R. O'Brien. Drag Me, Drop Me, Treat Me Like an Object. In *Proceedings of CHI'95: Human Factors in Computing Systems*, pages 525–530, 1995.
16. R. Bundgen. Reduce the Redex → ReDuX. In *Rewriting Techniques and Applications*, LNCS 690, pages 446–450. Springer, 1993.
17. N. Kawaguchi, T. Sakabe, and Y. Inagaki. TERSE: TErm Rewriting Support Environment. In *Workshop on ML and its Application*, pages 91–100, florida, june 1994. ACM SIGPLAN.

18. P. Viry. A user-interface for Knuth-Bendix completion. In *4th Workshop on User Interfaces for Theorem Provers (UITP'98)*, July 1998.
19. R. Saracco, J. Smith, and R. Reed. *Telecommunications Systems Engineering using SDL.* North-Holland, Elsevier Science Publishers, Amsterdam, 1989.
20. E. Najm (ed.). G-LOTOS: DAM1 to ISO8807 on graphical representation for LOTOS. Technical report, ISO/IEC JTC 1 / SC 21 N. 4871, 1992.
21. T.Bolognesi and D.Latella. Techniques for the formal definition of the G-LOTOS syntax. In *Procceeing of the 1987 IEEE Workshop on Visual Languages (VL'89)*, Roma, 1987.
22. J. L. Peterson. *Petri Net Theory and The Modeling of Systems.* Prentice-Hall, 1981.
23. M. M. Burnett and A. L. Ambler. A Declarative Approach to Event-handling in Visual Programming Languages. In *Proceedings of the 1992 IEEE Workshop Visual Languages (VL'92)*, pages 34–40, Seattle, Washington, September 1992.
24. B. Bell and C. Lewis. ChemTrains: A Language for Creating Behaving Pictures. In *Proceedings of the 1993 IEEE Symposium Visual Languages (VL'93)*, pages 188–195, Bergen, Norway, August 1993.
25. B. Shneiderman. *Designing the User Interface (Third Edition).* Addison-Wesley Publishing Company, 1997.

Chapter 8

On Extracting Algebraic Specifications from Untyped Object-Oriented Programs

Hirotaka Ohkubo [a] Toshiki Sakabe and Yasuyoshi Inagaki [b]

[a]Faculty of Information Science and Technology, Aichi Prefectural University, Nagakute, Aichi 480-1198, Japan

[b]Graduate School of Engineering, Nagoya University, Nagoya 464-8603, Japan

We present a transformation from object-oriented programs to order-sorted algebraic specifications. The transformation can be used for proving that an object-oriented program is a correct implementation of an algebraic specification. This work is an extension of Hennicker and Schmitz [3] in the following sense. The object-oriented language has essential object-oriented features such as polymorphism and inheritance, and the specification language is an order-sorted one, while a typed object-oriented language without inheritance and an ordinary many sorted algebraic specification language are used in [3].

1. Introduction

Hennicker and Schmitz [3] proposed a framework for proving correctness of object-oriented programs. That is, it gave a transformation from object-oriented programs to algebraic specifications, formalized the concept of observational implementation, and presented a method for proving that the algebraic specification obtained by the transformation is an observational implementation of an algebraic specification. The object-oriented language used in [3] is naively typed, and excludes the two essential features of object-oriented programs, namely inheritance and polymorphism.

In this paper, we introduce polymorphism and class inheritance into the framework of [3], and propose a new transformation. Our transformation first uses the type inference algorithm proposed by Ohkubo et al. [4,5] for untyped object-oriented languages. The type inference algorithm assigns a class set type, i.e., a set of class names, to each expression in a given object-oriented program. Thus object-oriented programs are made typed by the type inference algorithm. Then, by regarding the assigned class sets as the sorts, an algebraic specification is constructed from the object-oriented program according to the idea of [3], where order-sortedness is used in connection with polymorphism.

This paper is organized as follows: In section 2 we overview the framework of [3] and the type inference of the object-oriented language TinyObject. In section 3, we show the new transformation algorithm from TinyObject to order-sorted algebraic specifications. We give, as an example, a TinyObject program and the specification generated by our transformation. In section 4, we discuss the relationship between the algebraic semantics of TinyObject given in this paper and the operational semantics of TinyObject presented

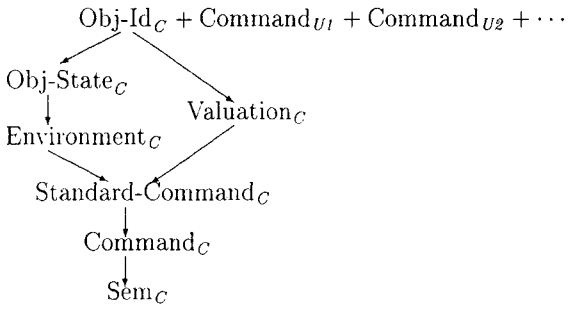


Figure 8.1. The hierarchical structure of the algebraic specification Sem_C

in [5].

2. Overview of Hennicker & Schmitz's Transformation and Type Inference of TinyObject

2.1. Hennicker & Schmitz's Transformation

In this section we overview the transformation proposed in [3], which we call H-S transformation in the sequel. H-S transformation produces from a given object-oriented program the algebraic specification of the meaning of the program, conforming to the standard denotational semantics of imperative programming languages.

The source language of H-S transformation is a subset of the object-oriented programming language OP proposed in [1]. The subset language excludes inheritance and polymorphism. The target language is the algebraic specification language of [6].

For a given source program P, H-S transformation constructs an algebraic specification Sem_C for each class C in P. Sem_C has the hierarchical structure illustrated in figure 8.1, where it is assumed that C uses classes $U_1, U_2, \ldots$ and the algebraic specification $\mathrm{Command}_{U_i}$ are already constructed from U_i by the transformation. Thus, H-S transformation constructs the whole specification of P traversing the tree of "has-a" or "use" relation over the classes of P from the leaves to the root. The following is a brief explanation of what each of the components of Sem_C specifies.

- Obj-Id$_C$ is a specification of the object identities of a class C. It defines a sort "$\mathtt{id}_C$" and a constant "$\mathtt{void}_C :\!\rightarrow \mathtt{id}_C$" to be the set of all object identities of C and the void object of C, respectively. This specification includes Command$_U$ if C uses class U.

- Obj-State$_C$ is an extension of Obj-Id$_C$. It specifies the states of objects of C as records with one field for each attribute of C. The sort of the states is "$\mathtt{state}_C$". The operators are "$\mathtt{update\_}a_C : \mathtt{state}_C, T \rightarrow \mathtt{state}_C$" and "$\mathtt{lookup\_}a_C : \mathtt{state}_C \rightarrow T$"

for each attribute a. and $\texttt{init}_C$, where T is a basic data type or $\texttt{id}_U$ determined by the class of the attribute a.

- Environment$_C$ extends Obj-State$_C$ to specify environments which are mappings that associate object identities of C with object states of C. The sort of the environments is `env`.

- Valuation$_C$ is an extension of Obj-Id$_C$. It specifies valuations of program variables which are mappings from program variables to basic data or object identities. The sort of the valuations is `valuation`.

- Standard-Command$_C$ extends Environment$_C$ and Valuation$_C$ to specify the state transition caused by the standard commands of the language in terms of the operation "$\texttt{exec}_s$: $\texttt{com}_s, \texttt{valuation}, \texttt{env} \rightarrow \texttt{env}, \texttt{id}_s$", where s is a class of P, $\texttt{com}_s$ is the sort of commands which return an object of class s, $\texttt{exec}_s$ changes the given environment under the valuation according to the command and delivers an object identity as the result.

  For example, the meaning of the sequencing command ";" is defined by the conditional equations

$$\texttt{exec}_{s_2}(t_1;_{s_1,s_2} t_2, v, p) = \texttt{exec}_{s_2}(t_2, v, p') \text{ if } \texttt{exec}_{s_1}(t_1, v, p) = (p', x_1)$$

  for all classes $s_1, s_2 \in \{C, U_1, \ldots, U_n\}$. The meaning of other commands such as assignment and conditional branching is defined in a similar way.

- Command$_C$ is Standard-Command$_C$ extended by adding those axioms for the operation `exec` which specify the meaning of method calls for the methods declared in C.

  For example, let m be a method in class C with a parameter X_1 of class U. Then Command$_C$ has the following conditional equation.

$$\begin{aligned} & \texttt{exec}_E(m(t_0, t_1), v, p) \\ = \; & \texttt{exec}_E(\langle \text{method body of } m \text{ in } C \rangle, v(\texttt{self}_C \rightarrow_C x_0)(X_1 \rightarrow_U x_1), p_1) \\ & \quad \text{if } \texttt{exec}_C(t_0, v, p) = (p_0, x_0) \text{ and } \texttt{exec}_U(t_1, v, p_0) = (p_1, x_1) \end{aligned}$$

  where E is the class of the object returned by m.

- Sem$_C$ extends Command$_C$ with declarations of class names as sorts and methods as operators. The axioms of Sem$_C$ convert the terms consisting of these operators into "`exec`" operator terms, which are computable by the axioms defined so far.

It is obvious that a specification can not be constructed from a program if the classes of the returned values of all commands are not determined without executing the program. Thus, H-S transformation assumes that all programs satisfy the condition that for any expression E the class of the value obtained by evaluating E is statically determined.

```
program                          ::= class-desc* class-desc
class-desc                       ::= class class name
                                     [inherits parent class name ]
                                     variable* method*
method                           ::= method selector exp
exp                              ::= void
                                   | self
                                   | arg
                                   | variable
                                   | myClass new
                                   | class name new
                                   | exp ; exp
                                   | if exp then exp else exp fi
                                   | variable := exp
                                   | exp <=selector (exp)
                                   | inherited selector (exp)
class name,variable,selector ::= identifier
```

Figure 8.2. Syntax of TinyObject

```
class N
class A
  method n N new
class B
  method n self <= o(arg)
  method o N new
class C
  v
  method m
    v := if arg then A new else B new;
    v <= n(void)
```

Figure 8.3. An example program of TinyObject

2.2. TinyObject and its Type Inference

TinyObject was proposed in [4,5] as the underlying language for detection of "Message Not Understood" errors in object-oriented programs using the inference algorithm for class set types. In this section, we review TinyObject and its type inference by examples.

TinyObject is an untyped imperative object-oriented language with inheritance. The syntax is given in figure 8.2. An example of TinyObject program is given in figure 8.3.

The example program in figure 8.3 has four classes N,A,B,C. In the class C, a method m is declared to have the body in which first an instance of class A or B is generated depending on the value of arg and assigned to the instance variable v, and then a message of selector n and the void object as argument is sent to it. Here arg is a reserved word which denotes one and only one argument common to all methods. When executing the message sending statement, the class which the method n to be called belongs to is determined depending on the class of v. This is an instance of the polymorphism common to standard object-oriented languages. Here, the message sending statement will not cause "Message Not Understood" error since class A and B both have method n.

A type inference algorithm was proposed for TinyObject although TinyObject is not a typed language. For a given program this algorithm assigns a class set type, i.e., a set of class names to each expression occurring in the program. It was shown that if the algorithm succeeds in type inference for a given program then no "Message Not Understood" error occurs during any executions of the program.

Let us apply the type inference algorithm to the above TinyObject program. The resulting type assignment $\langle f_{expr}, f_{var}, f_{arg} \rangle$ is shown in figure 8.4. f_{expr} is shown by dashed lines with sets attached, like `-----{...}`. That is, each expression is underlined with a dashed line and the set attached to the end of the dashed line is the class set type which is inferred by the algorithm for the expression. ALL stands for {A,B,C,N}. f_{var} and f_{arg} determine the class set type of each declaration of an instance variable and the argument of each method, which are indicated by `^^^^^` and `=====`.

An outline of the type inference algorithm is given in Appendix.

3. Transformation of Polymorphic TinyObject Programs to Order-sorted Algebraic Specifications

In this section, we show a transformation from TinyObject to CafeOBJ. Our transformation consists of two phases: the type inference phase and the construction phase of algebraic specifications.

In the type inference phase, we use the class set type inference illustrated in the previous section to assign a class set type to each expression of the source program. Those types are used as sorts in the target algebraic specification.

We explain the construction phase by using the example program for which the type assignment is inferred as shown in figure 8.4.

(1) First specify the void object.

```
[ id-void ]
op Void : -> id-void
```

(2) Declare a sort for each class set type that is assigned to some expression of P and for each class set type consisting of a single class name in P. Besides these sorts, declare one more sort id-ALL which corresponds to the class set type consisting of all the class names in P. If one class set type is a subset of another then the subsort

```
class N
class A
  method n           arg : ALL
    N new            =========
    -----{N}
class B
  method n           arg : ALL
    self <= o(arg)   =========
    ----{B}   ---ALL
    --------------{N}
  method o           arg : ALL
    N new            =========
    -----{N}
class C
  v : {A,B}
  ^^^^^^^^^
  method m           arg : ALL
                     =========
    v := if arg then A new else B new;
             ---ALL   -----{A}   -----{B}
         ----------------------------{A,B}
    -------------------------------{A,B}
    v <= n(void)
    -{A,B} ----{}
    -----------{N}
```

Figure 8.4. Result of type inference

relation is declared between the sorts corresponding to those class set types. The subsort relation is declared between the sort `id-void` and all the other sorts. The last subsort relation reflects the fact that `void` is a constant which can appear in any context during execution.

```
[ id-void < id-N ]
[ id-void < id-A ]
[ id-void < id-B ]
[ id-void < id-C ]
[ id-void id-A id-B < id-AB ]
[ id-void id-N id-A id-B id-C id-AB < id-ALL ]
```

(3) For each class C of P declare a sort of record `state-`C which has one field for each instance variable of C. The sorts of the fields are determined from f_{var} of the result of class set type inference. This record type is equipped with the initial state function in addition to the access functions provided by CafeOBJ.

```
record state-C { v : id-AB }
op init-C : -> state-C
eq init-C = state-C { v = Void } .
```

(4) Specify environments as mappings from objects to the states specified above.

`new-C` creates a new object which is not in the domain of `env`, associates it with the initial state `init-C`, and returns the pair of the updated `env` and the new object. `envset-v` and `envget-v` are the accessing functions to environments.

```
[ env ]
[ state-N state-A state-B state-C < state-ALL ]
op empty-env : -> env
op _ [ _ to _ ] : env id-ALL state-ALL -> env
op _ [ _ ] : env id-ALL -> state-ALL
op
axioms {
  var p : env
  vars i,j : id-ALL
  var s : state-ALL
  eq p [ i to s ] [ i ] = s .
  cq p [ i to s ] [ j ] = p [ j ] if i =/= j .

}
```

```
-- for instance variable v of class C
  op envset-v : env id-C id-AB -> env
  op envget-v : env id-C -> id-AB
  axioms {
    var p : env
    var i : id-C
    var j : id-AB
    var s : state-ALL
    eq envset-v(p,i,j) = p [ i to set-v(p[i],j)]
    eq envget-v(p,i) = v(p[i])
  }
-- instance creation
  op new-N : env -> result
  op new-A : env -> result
  op new-B : env -> result
  op new-C : env -> result
```

where `set-v` and `v` are the access functions of `state-C`, `result` is declared later as the direct product of `env` and `id-ALL`, and `state-ALL` is an auxiliary sort which is declared to be greatest among `state-`C and thereby avoids duplicating almost the same specifications of `env`.

(5) Declare a sort `val` as a record. This sort denotes the set of all valuations. A valuation is a mapping from program variables { `self, arg` } to objects.

```
record val { self : id-ALL, arg : id-ALL }
```

(6) Finally, specify the function `exec : com val env -> result`, by which the meaning of expressions is defined. The sort `com` is intended to denote the set of all expressions of TinyObject. This specification is separated into three parts.

(6-1) The first part defines the meaning of "`void`", "`self`", "`arg`", "`myClass new`", "C `env`" and "`if`$\cdots$`then`$\cdots$`else`$\cdots$`fi`" in a standard manner. For example,

```
record result { env : env id : id-ALL }

signature {
  op exec : com val env -> result

  op _;_ : com com -> com
}
axioms {
  vars t1 t2 : com
```

```
  var va : val
  var p : env

  eq exec(t1 ; t2,va,p) = exec(t2,va,env(exec(t1,va,p))) .
}
```

This part is common to all source programs.

(6-2) The second part specifies the meaning of those expressions which access instance variables. For each variable v, add two expressions of reading and writing the variable, into com. and declare evaluations of those expressions.

```
op v : -> com
op v:=_ : com -> com

axioms {
  var t : com
  eq exec(v,va,p) =
    result { env = p, id = envget-v(self(va)) } .
  let res-t = exec(t,va,p) .
  eq exec(v:= t,va,p) =
    result { env = envset-v(env(res-t),self(va),id(res-t)),
             id  = id(res-t) } .
```

(6-3) The third part specifies the meaning of message sending expressions. For each method of selector m, declare an operator m which gives the method body com, and define an equation of exec concerning method call expressions.

```
op n : id-ALL -> com
op o : id-ALL -> com
op m : id-ALL -> com
axioms {
  var a : id-A
  var b : id-B
  var c : id-C

  eq n(a) = N new .
  eq n(b) = self <= o (arg) .
  eq o(b) = N new .
  eq m(c) = v := if arg then A new else B new; v <= n(void) .
}

axioms {
  vars R,A : com
```

```
    var va : val
    vars p,p' : env

    let r = exec(R,v,p) .
    let a = exec(A,v,env(exec(R,v,p))) .
    eq exec(R <= n(A),v,p) =
       exec(n(r), set-arg(set-self(v,id(r)),id(a)), env(a)) .
    eq exec(R <= o(A),v,p) =
       exec(o(r), set-arg(set-self(v,id(r)),id(a)), env(a)) .
    eq exec(R <= m(A),v,p) =
       exec(m(r), set-arg(set-self(v,id(r)),id(a)), env(a)) .
}
```

In the example program, the two expressions of the method `m` of the class `C` are both polymorphic. In the first expression, the class of the object assigned to variable `v` varies depending on the condition. H-S transformation can not handle such a polymorphic expression. Our transformation handles it by introducing a supersort `id-AB`.

The second expression includes so called dynamic dispatch. That is, the method body actually invoked varies depending on the class of the receiver object. H-S transformation cannot handle such dynamic dispatch. Our transformation works well for expressions including dynamic dispatch owing to order-sortedness again.

H-S transformation constructs the target algebraic specification class by class traversing the "has-a" relation tree from the leaves to the root. This means that if there is a "has-a" relation loop, i.e., a mutual "has-a" reference then H-S transformation does not work. On the other hand, our transformation constructs the whole specification for all classes at a time and hence works well even if there are mutual "has-a" references. Further this construction process is essential to handle polymorphism.

A class inheritance causes no problem in our transformation because of f_{var} of type assignment inferred by the class set type inference algorithm. However an "`inherited`" expression causes a problem. We can overcome this problem by renaming method names.

4. Relationship between Algebraic and Operational Semantics of TinyObject

An operational semantics of TinyObject was given in [5] as the semantic function from TinyObject programs to TinyObject Machines. Soundness of class set type inference was proved with respect to this semantics. The translation from TinyObject to order-sorted specifications given in this paper can be seen as an algebraic semantics of TinyObject. We discuss in this section how the two semantics of TinyObject are related.

4.1. TinyObject Machine

For any sets X and Y let $[X \to Y]$, $[X \rightsquigarrow Y]$ and 2^X denote the set of all total functions from X to Y, the set of all partial functions from X to Y and the power set of X, respectively.

$$
\begin{array}{lcl}
\langle TomL \rangle & ::= & \varepsilon \mid \langle Comm \rangle \langle TomL \rangle \\
\langle Comm \rangle & ::= & \text{void} \mid \text{self} \mid \text{arg} \\
 & & \mid \text{load}(\langle Vari \rangle) \mid \text{store}(\langle Vari \rangle) \\
 & & \mid \text{new}(\langle Class \rangle) \mid \text{new}() \\
 & & \mid \text{cond}(\langle TomL \rangle . \langle TomL \rangle) \\
 & & \mid \text{send}(\langle Mesg \rangle) \\
 & & \mid \text{inherited}(\langle Class \rangle . \langle Mesg \rangle) \\
 & & \mid \text{recover} \mid \text{pop} \\
\langle Class \rangle & ::= & c \\
\langle Vari \rangle & ::= & v \\
\langle Mesg \rangle & ::= & m
\end{array}
$$

where $c \in Cls$, $v \in Var$, $m \in Sel$.

Figure 8.5. Syntax of *TomL*

A TinyObject Machine (TOM, for short) is a state transition machine with two stacks, which is defined as a seven tuple $M = (Obj, Cls, Sel, Var, \Phi, \Psi, \perp)$, where

- $Obj = \{o_0, o_1, \ldots\}$ is a countably infinite set of objects,
- Cls is a finite set of class names,
- Sel is a finite set of selectors,
- Var is a finite set of variables,
- $\Phi \in [Cls \rightarrow [Sel \rightsquigarrow TomL]]$ assigns to a class a mapping which determines the method body of each selector in the class,
- $\Psi \in [Cls \rightarrow 2^{Var}]$ specifies the set of instance variables of a class, and
- $\perp$ is a special object not included in Obj, which is called the void object.

Here, *TomL* is the TOM language, the syntax of which is given in figure 8.5.

A state of the TOM is a six tuple $(C, V, self, arg, p, st)$, where

- $C \in [Obj \rightsquigarrow Cls]$
- $V \in [Obj \rightsquigarrow [Var \rightsquigarrow Obj \cup \{\perp\}]]$
- $self \in Obj$

- $arg \in Obj \cup \{\perp\}$
- $p \in TomL$, called instruction stack
- $st \in (Obj \cup \{\perp\})^*$, called evaluation stack

The last two components of a state are sequences working as stacks: the instruction stack is the fragment of *TomL* program to be executed and the evaluation stack is the object list for evaluation of expressions.

Execution of M is a (finite or infinite) sequence of states $q_0, q_1, \ldots$ such that q_0 is an initial state and $q_i \xrightarrow{step} q_{i+1}$ for any $i = 0, 1, 2 \ldots$, where $\xrightarrow{step}$ is a relation over the set of states which intuitively means the state transition caused by executing the top of the instruction stack. The definitions of the initial states and $\xrightarrow{step}$ are given in Appendix.

The TOM obtained from a TinyObject program P by the operational semantics is $M_P = (Obj, Classes_P, Selectors_P, Variables_P, \Phi_P, \Psi_P, \perp)$. Here, $Classes_P$, $Selectors_P$ and $Variables_P$ are the sets of class names, selector names and instance variables occurring in P, respectively. $\Phi_P(m)$ for each selector $m \in Selectors_P$ is the instruction sequence of the method body corresponding to m. For each class $c \in Classes_P$, $\Psi_P(c)$ is the set of instance variables of c. The definition of Φ_P is given in Appendix.

4.2. Relationship between Algebraic and Operational Semantics

Let P be a TinyObject program. Consider a term of the form $T = \texttt{exec}(E, v, env)$ of record sort `result { env : env id : id-ALL }` in the specification obtained from P, where E, v and env are terms of sort `com`, `val` and `env`, respectively. This term corresponds to the state of execution at the time when an occurrence of expression E in P is about to be executed. Let oc be the position of E in P and $[\![v]\!]$ and $[\![env]\!]$ be the valuation and the environment at that moment.

We define $\mathcal{I}(T)$ to be the state of M_P such that

$$\mathcal{I}(T) = (C, [\![env]\!], [\![v]\!](\texttt{self}), [\![v]\!](\texttt{arg}), Compile(oc), \varepsilon)$$

where C is a partial function defined by

$$C(oid) = \begin{cases} K & \text{if } [\![env]\!](oid) \in state_K \\ \text{undefined} & \text{otherwise} \end{cases}$$

and *Compile* is the function that is used in the definition of Φ_P (see Appendix).

For terms of the result record form, define

$$\mathcal{I}(\ \texttt{result \{ env =}e\texttt{, id = }o\texttt{\}}) = (C, [\![e]\!], [\![v]\!](\texttt{self}), [\![v]\!](\texttt{arg}), \varepsilon, [\![o]\!])$$

where C is as above.

Then we conjecture the following claim holds:

Claim Let T_1 be a term of the form $\texttt{exec}(E, v, env)$ and T_2 be any term of sort `result`. Then $T_1 \xrightarrow{\mathsf{Sem}_P} T_2$ if and only if $\mathcal{I}(T_1)(\xrightarrow{\mathsf{step}})^*\mathcal{I}(T_2)$, where $\xrightarrow{\mathsf{Sem}_P}$ is the one step reduction relation of the conditional term rewriting system that corresponds to the algebraic specification translated from P.

Let us examine this claim by example. Assume expression

$$E = \texttt{if arg then A new else B new fi}$$

at occurrence oc is about to be evaluated under the environment env and valuation v such that $v(\texttt{arg}) = v(\texttt{self})$ in the example program in figure 8.3.

Let $T_I = \texttt{exec}(E, v, env)$, then $\mathcal{I}(T_I) = (C, [\![env]\!], [\![v]\!](\texttt{self}), [\![v]\!](\texttt{arg}), \Phi(oc), \varepsilon)$, where $\Phi(oc) = \text{arg} : \text{cond}(\text{ new(A), new(B)})$.

By using the conditional equation

```
ceq exec(if E then Et else Ef fi,v,p)
  = exec(Et,v,env(exec(E,v,p)))
  if id(exec(E,v,p)) =/= void .
```

as a conditional rewriting rule we obtain

$$\begin{aligned} T_I &\xrightarrow{\text{Sem}_P} \texttt{exec(new A,}\, v, env)(= T_2) \\ &\xrightarrow{\text{Sem}_P} \texttt{result \{ env = }\, env\, \texttt{[}a_I\, \texttt{to initA], id = }\, a_I\}(= T_3) \end{aligned}$$

where a_I is an object of sort `id-A`. On the other hand we have

$$\begin{aligned} \mathcal{I}(T_I) &\xrightarrow{\text{step}} (C, [\![env]\!], [\![v]\!](\texttt{self}), [\![v]\!](\texttt{arg}), \text{cond(}\ \text{new(A), new(B)}\ \text{)}, [\![v]\!](\texttt{arg})) \\ &\xrightarrow{\text{step}} (C, [\![env]\!], [\![v]\!](\texttt{self}), [\![v]\!](\texttt{arg}), \text{new(A)}, \varepsilon)(= S_2) \\ &\xrightarrow{\text{step}} (C[A/a_I], [\![env]\!][\emptyset/a_I], [\![v]\!](\texttt{self}), [\![v]\!](\texttt{arg}), \varepsilon, a_I)(= S_3) \end{aligned}$$

where a_I is a new object of class A. ($f[b/a]$ stands for the updated partial function of f as $f[b/a](x) = b$ if $x = a$ and $f[b/a](x) = f(x)$ otherwise.) It is clear that

$$\begin{aligned} \mathcal{I}(T_2) &= S_2 \\ \mathcal{I}(T_3) &= S_3 \end{aligned}$$

Thus, the claim holds in this case.

To complete the proof of the claim, we will need to do the same check as above for all cases of expressions of TinyObject and may have to modify the definition of $\mathcal{I}(T)$ so that consistency of stacks is maintained during message sending. This is left as a future work.

5. Concluding Remarks

In this paper we have shown a transformation from TinyObject, an untyped object-oriented language, to order-sorted algebraic specifications. This transformation makes it possible to prove that an object-oriented program is an object-oriented implementation of an algebraic specification. We have experimentally implemented this transformation in Standard-ML [2]. This program takes a TinyObject program and returns a specification in CafeOBJ syntax. It is future work to develop proof techniques for the correctness of observational implementation in our framework.

REFERENCES

1. Breu,R: Algebraic Specification Techniques in Object Oriented Programming Environments, Springer Lecture Notes in Computer Science 562, 1991.
2. Eba, M.: On Algebraic Specification of Untyped Object-Oriented Programs, Master Thesis, Nagoya University, 1999.
3. Hennicker, R. and Schmitz, C.: Object-Oriented Implementation of Abstract Data Type Specifications, *AMAST96 Proc. (LNCS 1101)*, 1996, pp.163-179.
4. Ohkubo, H., Sakabe, T. and Inagaki, Y.: Soundness of Class Set Type Checking Method for Object-Oriented Programs, *Technical Report of IEICE*, SS95-13. 1995-07, pp. 23-30.
5. Ohkubo, H., Sakabe, T. and Inagaki, Y.: A Type Checking Algorithm for Detecting Message Not Understood Fault of Object Oriented Programs, to appear in Computer Software (in Japanese).
6. Wirsing, M.: Structured algebraic specifications : a kernel language, *Theoretical Computer Science*, 42, pp. 123-249, 1986.

A. Definitions

In this appendix we give the definitions needed in this paper. See [5] for details of these definitions.

A.1. The State Transition Relation $\xrightarrow{step}$ of TOMs

For any partial function $f : X \rightsquigarrow Y$ the updated partial function is denoted by $f[b/a]$. That is, $f[b/a](x) = b$ if $x = a$ and $f[b/a](x) = f(x)$ otherwise.

Let $M = (Obj, Cls, Sel, Var, \Phi, \Psi, \perp)$ be a TOM.

For each $c \in Cls$ and $m \in Sel$ the initial state $q_{init}(c, m)$ is defined by

$$q_{init}(c, m) = (\{(o_0, c)\}, \{(o_0, \{(v, \perp) \mid v \in \Psi(c)\})\}, o_0, \perp, \varepsilon, \Phi(c)(m))$$

The state transition relation $\xrightarrow{step}$ is defined as in table 8.1. Here ":" denotes the concatenation of sequences and $new(dom(C))$ is the object whose index is the least in $Obj - dom(C)$.

A.2. Definition of Φ_P

For any TinyObject program P define $\Phi_P : Classes_P \to [Selectors_P \rightsquigarrow TomL]$ as follows: For any $c \in Classes_P$ and $m \in Selectors_P$,

$$\Phi_P(c)(m) = \begin{cases} Compile(c, E) & \text{if a method is defined for selector } m \text{ in class } c \text{ and the method body is } E \\ \Phi_P(c')(m) & \text{if no method is defined for } m \text{ in } c \text{ and } Inh_P(c) = c' \\ \text{undefined} & \text{otherwise} \end{cases}$$

Here, $Inh_P(c)$ is the class that is inherited by c in P and $Compile$ is the function defined in table 8.2.

$$(C, V, s, a, st, \text{void} : I) \xrightarrow{step} (C, V, s, a, \bot : st, I)$$
$$(C, V, s, a, st, \text{self} : I) \xrightarrow{step} (C, V, s, a, s : st, I)$$
$$(C, V, s, a, st, \text{arg} : I) \xrightarrow{step} (C, V, s, a, a : st, I)$$
$$(C, V, s, a, st, \text{load}(v) : I) \xrightarrow{step} (C, V, s, a, V(s)(v) : st, I)$$
$$(C, V, s, a, o : st, \text{store}(v) : I) \xrightarrow{step} (C, V[V(s)[o/v]/s], s, a, o : st, I)$$
$$(C, V, s, a, st, \text{new}() : I) \xrightarrow{step} (C[C(s)/o], V[\{(v, \bot) | v \in \Psi(C(s))\}/o], s, a, o : st, I) \quad \text{where } o = new(dom(C))$$
$$(C, V, s, a, st, \text{new}(c) : I) \xrightarrow{step} (C[c/o], V[\{(v, \bot) | v \in \Psi(c)\}/o], s, a, o : st, I) \quad \text{where } o = new(dom(C))$$
$$(C, V, s, a, \bot : st, \text{cond}(I_1, I_2) : I) \xrightarrow{step} (C, V, s, a, st, I_2 : I)$$
$$(C, V, s, a, o : st, \text{cond}(I_1, I_2) : I) \xrightarrow{step} (C, V, s, a, st, I_1 : I) \quad \text{where } o \neq \bot$$
$$(C, V, s, a, a' : s' : st, \text{send}(m) : I) \xrightarrow{step} (C, V, s', a', a : s : st, \Phi(C(s'))(m) : I)$$
$$(C, V, s, a, a' : s : st, \text{inherited}(c, m) : I) \xrightarrow{step} (C, V, s, a', a : s : st, \Phi(c)(m) : I)$$
$$(C, V, s', a', result : a : s : st, \text{recover} : I) \xrightarrow{step} (C, V, s, a, result : st, I)$$
$$(C, V, s, a, k : st, \text{pop} : I) \xrightarrow{step} (C, V, s, a, st, I)$$

Table 8.1
State transition of TinyObject Machines

Expression E	$Compile(c, E) \in TomL$
`void`	void
`self`	self
`arg`	arg
v	load (v)
`myClass new`	new()
d `new`	new(d)
`if` *E1* `then` *E2* `else` *E3*	*Compile*(c,*E1*) : cond (*Compile*(c,*E2*), *Compile*(c,*E3*))
E1 ; *E2*	*Compile*(c,*E1*) : pop : *Compile*(c,*E2*)
v `:=` *E1*	*Compile*(c,*E1*) : store(v)
Er `<=`m`(`*Ea*`)`	*Compile*(c,*Er*) : *Compile*(c,*Ea*) : send(m) : recover
`inherited` m`(`*Ea*`)`	self : *Compile*(c,*Ea*) : inherited ($Inh_P(c), m$) : recover

Table 8.2
Definition of *Compile*

A.3. Class Set Type Inference

Type Assignments

Let P be a TinyObject program. A triple of partial functions $\langle f_{expr} : Classes_P \times Occ_P \rightsquigarrow 2^{Classes_P}, f_{arg} : Classes_P \times Selectors_P \rightsquigarrow 2^{Classes_P}, f_{var} : Classes_P \times Variables_P \rightsquigarrow 2^{Classes_P} \rangle$ is called a type assignment of P, where $Classes_P$ is the set of all class names, Occ_P is the set of all occurrences of expressions, $Selectors_P$ is the set of all selectors, and $Variables_P$ is the set of all variables in P.

A type assignment $\langle f_{expr}, f_{arg}, f_{var} \rangle$ of P is said to be correct if the following three conditions hold:

(1) If an expression at occurrence oc in P is evaluated to an object in class d under an environment such that *self* belongs to c then $d \in f_{expr}(c, oc)$.

(2) If selector m is sent to an object in class c and the argument is an object in class d then $d \in f_{arg}(c, m)$.

(3) If instance variable v of an object in class c has an object in class d then $d \in f_{var}(c, v)$.

Type Constraints

Define type terms as follows. Let $Classes_P = \{c_1, c_2, \cdots, c_k\}$.

1. Constants: $\{\}, \{\Box\}$ and $\{c\}$ for $c \in Classes_P$ are type terms.
2. Type variables: $\mathrm{Xexpr}_{c,d,oc}$, $\mathrm{Xarg}_{c,d,m}$, $\mathrm{Xret}_{c,d,m}$, $\mathrm{Xvar}_{c,v}$ are type terms, where $c \in Classes_P \cup \{\Box\}$, $d \in Classes_P$, $m \in Selectors_P$, $oc \in Occ_P$, v is an instance variables occurring in P.
3. Composite type terms: If $T, T_1, T_2, \cdots, T_k$ are type terms then
   (1) $T_1 \cup T_2$, $T_1 \cap T_2$ are type terms,
   (2) $\sqcap(T, T_1, T_2, \cdots, T_k)$ is a type term if $T \neq \{\}$, which is interpreted as $\bigcap_{c_i \in T} T_i$
   (3) $\sqcup(T, T_1, T_2, \cdots, T_k)$ is a type term if $T \neq \{\}$, which is interpreted as $\bigcup_{c_i \in T} T_i$

The following abbreviations are used later:

- HAS_m for $\{c_{i1}\} \cup \cdots \cup \{c_{in}\}$ if $c_{i1}, c_{i2}, \cdots, c_{in}$ is the set of all classes that have a method for selector m
- $\mathrm{domain}_m(T)$ for $\sqcap(T, \mathrm{Xarg}_{c_1,c_1,m}, \cdots, \mathrm{Xarg}_{c_k,c_k,m})$
- $\mathrm{range}_m(T)$ for $\sqcup(T, \mathrm{Xret}_{c_1,c_1,m}, \cdots, \mathrm{Xret}_{c_k,c_k,m})$

For any type terms T_1 and T_2 the following two formulas are type constraints:

$$T_1 = T_2$$
$$T_1 \subseteq T_2$$

$$CC(c,m) = \begin{cases} \{\mathrm{Xret}_{\square,c,m} = \mathrm{Xexpr}_{\square,c,oc}\} \cup CF(c,m,oc) \cup CC(p,m) \\ \quad \text{if the body of } c \text{ has "}\texttt{method}\ m\ \underline{\mathrm{exp}}\text{" at position } oc \text{ and} \\ \quad \text{"}\texttt{inherited}\ m\text{" at some position and } p = Inh_P(c) \\ \{\mathrm{Xret}_{\square,c,m} = \mathrm{Xexpr}_{\square,c,oc}\} \cup CF(c,m,oc) \\ \quad \text{if the body of } c \text{ has "}\texttt{method}\ m\ \underline{\mathrm{exp}}\text{" at position } oc \text{ and} \\ \quad \text{no "}\texttt{inherited}\ m\text{"} \\ \{\mathrm{Xarg}_{\square,c,m} = \mathrm{Xarg}_{\square,p,m}, \mathrm{Xret}_{\square,c,m} = \mathrm{Xret}_{\square,p,m}\} \cup CC(p,m) \\ \quad \text{if the body of } c \text{ has no "}\texttt{method}\ m\ \underline{\mathrm{exp}}\text{" and } p = Inh_P(c) \\ \{\} \quad \text{otherwise} \end{cases}$$

Figure 8.6. Definition of CC

Class Set Type Inference

Outline of the type inference algorithm is as follows:

Input: A TinyObject program P

Output: A correct type assignment of P if exists, "Type error" otherwise

Method:

(1) Construct $CC(c,m)$ for each $c \in Classes_P$ and $m \in Selectors_P$. The definition of $CC(c,m)$ is in figure 8.6. $CC(c,m)$ is the set of constraint formulas directly obtained from the source code of method m of class c. It contains holes($\square$) to be replaced later with proper class names.

(2) Construct PC by

$$PC = \bigcup_{c \in Classes_P} \bigcup_{m \in Selectors_P} (CC(c,m)[c/\square])$$

where $CC(c,m)[c/\square]$ is the type constraint obtained from $CC(c,m)$ by replacing $\square$ with c. PC is the set of constraint formulas for program P.

(3) Solve the type constraints PC. If PC has a solution, then

$$\begin{cases} f_{expr}(c,oc) &= \text{(solution of } \mathrm{Xexpr}_{c,c,oc}) \\ f_{arg}(c,m) &= \text{(solution of } \mathrm{Xarg}_{c,c,m}) \\ f_{var}(c,v) &= \text{(solution of } \mathrm{Xvar}_{c,v}) \end{cases}$$

is the type assignment for P.

Expression E_{oc}	Set of constraints $CF(c, m, oc)$
`void`	$\{\text{Xexpr}_{\square,c,oc} = \{\}\}$
`self`	$\{\text{Xexpr}_{\square,c,oc} = \{\square\}\}$
`arg`	$\{\text{Xexpr}_{\square,c,oc} = \text{Xarg}_{\square,c,m}\}$
v	$\{\text{Xexpr}_{\square,c,oc} = \text{Xvar}_{\square,v}\}$
`myClass new`	$\{\text{Xexpr}_{\square,c,oc} = \{\square\}\}$
d `new`	$\{\text{Xexpr}_{\square,c,oc} = \{d\}\}$
{ $E_{oc:1}$; $E_{oc:2}$ }	$\{\text{Xexpr}_{\square,c,oc} = \text{Xexpr}_{\square,c,oc:2}\}$ $\cup CF(c, m, oc:1) \cup CF(c, m, oc:2)$
`if` $E_{oc:1}$ `then` $E_{oc:2}$ `else` $E_{oc:3}$ `fi`	$\{\text{Xexpr}_{\square,c,oc} = \text{Xexpr}_{\square,c,oc:2} \cup \text{Xexpr}_{\square,c,oc:3}\}$ $\cup CF(c, m, oc:1) \cup CF(c, m, oc:2) \cup CF(c, m, oc:3)$
$v := E_{oc:1}$	$\{\text{Xvar}_{\square,v} \supseteq \text{Xexpr}_{\square,c,oc:1}, \text{Xexpr}_{\square,c,oc} = \text{Xexpr}_{\square,c,oc:1}\}$ $\cup CF(c, m, oc:1)$
$E_{oc:1}$ `<=`$t(E_{oc:2})$	$\{\text{Xexpr}_{\square,c,oc:1} \subseteq \text{HAS}_t,$ $\text{domain}_t(\text{Xexpr}_{\square,c,oc:1}) \supseteq \text{Xexpr}_{\square,c,oc:2},$ $\text{Xexpr}_{\square,c,oc} = \text{range}_t(\text{Xexpr}_{\square,c,oc:1})\}$ $\cup CF(c, m, oc:1) \cup CF(c, m, oc:2)$
`inherited` $t(E_{oc:1})$	$\{\text{Xarg}_{\square,Inh_P(c),t} \supseteq \text{Xexpr}_{\square,c,oc:1}, \text{Xexpr}_{\square,c,oc} = \text{Xret}_{\square,Inh_P(c),t}\}$ $\cup CF(c, m, oc:1)$

Table 8.3
Definition of CF

Chapter 9

An Environment for Systematic Development of Algebraic Specifications on Networks

Akishi Seo [a] and Ataru T. Nakagawa [b]

[a]Nihon Unisys, Ltd.
1-1-1 Toyosu, Koto-ku, Tokyo 135-8560, Japan
E-mail: Akishi.Seo@unisys.co.jp

[b]SRA Software Engineering Laboratory
Marusho Bld. 5F, 3-12 Yotsuya, Shinjuku-ku, Tokyo 160-0004, Japan
E-mail: nakagawa@sra.co.jp

In this paper we propose a framework that enables users to develop algebraic specifications in a distributed environment as if they were working on a single system. They need not aware of where all those modules and processors are located, and are able to concentrate on the immediate technical problems at hand; at the same time all the resources in the enviroment are at their disposal. This framework is an application of "Computing as Editing" paradigm and exploits current web technologies. We implemented an integrated software development environment that is based on the framework. In this environment, users can develop CafeOBJ specifications on local, organisational, or even worldwide networks, through standard tools such as Emacs or Netscape.

1. Introduction

Development of specifications in the language CafeOBJ[1] [6][1] is a process of constructing and verifying interrelated modules. Those modules are not necessarily written by one person: it is rather the case that they are written by many, as long as the target system is not small. In general, the larger the system, the more people is involved in specification development.

In such a situation, it is important, with regard to productivity and reliability, for all these members to understand outlines of specifications quickly. To add intuitive explanations or illustrations to formal specifications is an effective way to achieve that purpose, even if the extra information does not make any technical sense.

Moreover, in these days a specification developer usually uses more than one machines connected with each other, even if he is working in total isolation. This indulgence makes sense to the solitary developer, at least for two reasons. One is that it enables him to use

[1] CafeOBJ sometimes refers to a language and sometimes to a processor for the language. It may sound a little verbose, but throughout this paper we call the language "the language CafeOBJ" and the processor "the system CafeOBJ", unless it is too obvious which is referred to.

the full power of each machine. Another is that it allows him to pursue a single work in different places hopping from one place to another as he wishes.

Almost all of today's computer systems are able to connect with each other irrespective of their machine types or operating systems, thanks to the de facto standard protocols such as TCP/IP. Once the system of a developer is connected to a global network, such as the Internet, a huge amount of potential resources suddenly becomes at his disposal. He can use diverse platform environments without bothering with details about how to send/receive documents. It is no longer a severe problem, then, that a particular tool, such as a language processor or a prover, can work on a limited range of operating systems and environments. A developer simply uses his own machine for any purpose other than using that tool; only when a need arises, he connects to that machine which happens to be able to run that tool.

CafeOBJ — both the language and the system — is the one we are concerned here. In view of recent computer scenes as explained above, we envisioned a following situation.

- Several developers are writing and checking interrelated modules.
- Each of them is working on his own machine.
- Modules reside within the machines where they have been developed.
- The machines are running various operating systems. They are different models from different vendors.
- The system CafeOBJ is running on a machine.
- A prover for the language CafeOBJ is running on another machine.
- Another prover is running on yet another machine.
- Somehow, in sipte of all these diverse machines and locations, each developer is doing his work without any hindrance.

How can it happen? Comparing with the situation where all the developers are working on a centralised system on which all these tools are running, we took note of the the following problems to overcome.

0. At the basic level, these machines should be connected via networks. But this is too obvious and we simply assume so. And it is by now safe to assume that they are connected via a single network, in the logical sense.
1. Modules that have been developed may be distributed widely. How can a developer identify a specific module in the network?
2. Modules related with each other may be on separate machines. How are the relations between the modules maintained and resolved? Here the bare names of these modules are not enough, since there may be different modules of the same name.

3. Modules may be on machines on which the system CafeOBJ and/or provers are not running, or even cannot run. In such a case, how can a developer apply the tools to these modules?

We devised a framework[7] which deals with these problems (apart from that of physical connections) and enables you to develop algebraic specifications in a distributed environment as if you were working on a single system. In addition, the framework aims to make it easy for you to understand specifications in order to improve quality and productivity. We also implemented an integrated environment[2] that is based on the framework.

The rest of the paper is organised as follows. In the next section, we explain several desirable features of a specification development environment, setting our goal. Then we explicate the framework, called "Computing as Editing", in Section 3. In Section 4, we explain the integrated environment based on the framework. Section 5 summarises our conclusions and future works.

2. Specification Development Environment We Want

Before we begin to explicate the framework we rely on, we informally define a specification development environment that should be achieved by the framework.

2.1. Developing Specifications

Activities around specification development are diverse. Not only specifiers, but programmers, managers, and customers are involved at some stage. They have different interests and different perspectives, they have different competences and different tastes. To build a comprehensive environment for specifications, therefore, is a very complicated issue. We decided it too ambitious to accommodate all of these possibilities, and set our goal to helping activities of specifiers in the main.

The activities around specification development may be listed as: writing a specification, understanding one, and checking one. Our immediate target language is CafeOBJ, so to help these activities involves the following list of freatures. Our integrated specification development environment has to

- allow you to write codes. A dedicated editor was a possibility here, but we decided not to build one from scratch. We consider it best to leave it to you to choose whatever editor you like. The reason of this decision is the fact that no language-specific editor has become widely accepted throughout the entire history of software engineering. Our environment, in addition, has to

- allow you to add explanations in natural languages, graphics, tables, or even animations, as annotations to codes. They are very helpful in understanding specifications written in purely formal codes, and are especially useful if you have to cope with many modules written by others. Traditionally, these annotations appeared as "comments". We prefer to confer them a more enhanced status. They should be first-class citizens. Our environment, moreover, has to

- allow you to execute codes. This consideration is specific to the language CafeOBJ,

one of whose distinct features from the very beginning is executability. In addition, our environment has to

- allow you to state and prove properties of codes. The executability of the language CafeOBJ is in general not enough to prove useful properties, and additional provers, if available, are welcome. We have to provide an environment where they are usable seamlessly.
- Finally, all of those features should be available in a network transparent manner.

2.2. A System Overview

The framework that we propose has practical and reasonable assumptions on software and hardware, as follows.

- Client programs, the system CafeOBJ, provers and specifications may be put on machines that are connected with each other via a single network.
- A firewall may exist between a machine on which a client program runs and a machine on which the system CafeOBJ, a prover, and/or a specification is installed. This situation is common for configurations of commercial enterprises.
- An operating system of a client machine may be UNIX, LINUX, Macintosh, or Windows, or any other comparable system.
- As a client program, you may use standard tools, such as Emacs and Netscape.

In particular, the last assumption means that you need not get a special training in using frontend software. Figure 9.1 shows the system overview of the framework that is based on these assumptions.

3. On the Framework

We devised the framework to realise the specification development environment defined in the previous section. Relying on this framework, you can edit, display and search specifications distributed over a network, and can invoke the system CafeOBJ or a prover located somewhere in the network, without noticing you are surfing through the network.

3.1. Documents with Constraints

The framework is an application of "Computing as Editing" paradigm[4] to specification development. The framework also exploits web technologies so that users can develop specifications on worldwide networks.

"Computing as Editing" regards a computation as a document editing process. In this paradigm, the target of a computation is part of a document. The user defines a certain relation between parts of a document, and if a part does not satisfy such a relation, the part is revised accordingly.

A spreadsheet illustrates this idea well. Suppose there are several cells that contain individual values and one cell that contains their sum. In defining the latter as the cell

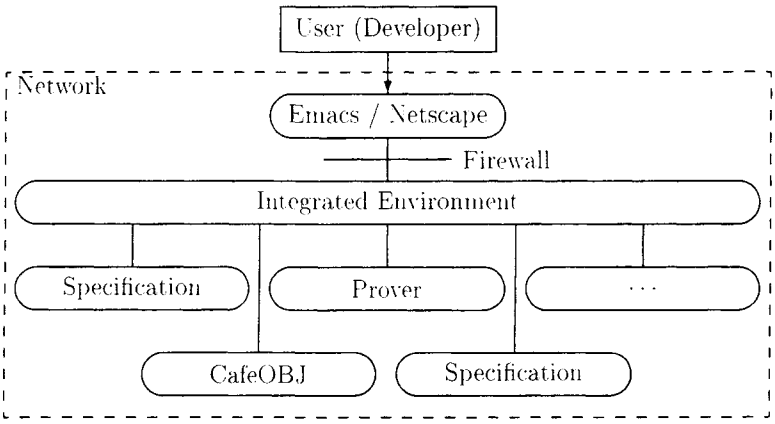


Figure 9.1. System Overview

for the sum of the values in the former, you impose a relation between these cells so that, whenever an individual value changes, the value of the sum cell have to reflect the change. This requires an (elementary school) computation.

Hagiya and Kakuno[4] generalised this observation to establish a computational paradigm where the essence of computations in a broad sense (including, for example, theorem proving) is captured as a process of document editing.

In case of spreadsheets the relations are usually in one direction: in the above example, the sum cell, and it alone, has to reflect changes in the other cells. In the general paradigm, this need not be the case: you may have a (nondeterministic) computation process where the change of the sum cell is reflected by the other cells. Even from this simple example, it is easily seen that this paradigm has a potential to incorporate a very wide spectrum of computations.

To put it a little more formally, the paradigm achieves computations by the following procedure.

1. Embed relations, called "constraints", between parts of documents. Constraints have to be satisfied when the documents are completed.

2. Alter a document so that one of the embedded relations becomes satisfied. To edit in such a way is called "to solve a constraint".

3. If all the constraints are satisfied, go to 4. Otherwise repeat 2.

4. The documents are completed.

The procedure to develop specifications in our framework, which is an application of the paradigm to specification development, is as follows:

1. Write specifications in the language CafeOBJ.
2. Specify the locations of imported modules if there are any.
3. Add some expositions in natural language to the specifications.
4. Add some graphics or animations to the specifications if necessary.
5. Write properties (theorems) the specifications must have, in a formal language. Usually they are equations of CafeOBJ[2].
6. Embed relations between parts of the specification documents as "constraints", which the documents will satisfy when they are completed. A constraint includes information about the system CafeOBJ or a prover to be used to solve the constraint.
7. Solve a constraint by the tool specified in the constraint.
8. Repeat solving a constraint until all constraints are solved.
9. The specifications are completed.

In this method, verification of properties of specifications goes on interactively by way of solving nested constraints repeatedly. Thus the user acquires the following flexibilities:

- A part of a specification may be associated with several constraints, and several parts of several specifications may be associated with a constraint.
- The order of solving constraints may be bottom up, top down, or their hybrid.
- The user can solve constraints step by step rather than all at once while he devises a strategy for a whole proof.

3.2. Forsdonnet

We developed a markup language, called "Forsdonnet"[3], to enable the user to describe specifications, to solve constraints, and to see specifications on networks. The language adds the following tags to HTML to extend the features of HTML. By the extension, the user can write a constrained specification as a hypertext and solve a constraint on worldwide networks.

`<MODULE>, </MODULE>`
: These are tags to specify a module or a view of the language CafeOBJ. A module or a view must be put between the tags[4].

[2] They may be disguised as a proof obligation. See the example in Section 3.3.
[3] FORmal Specification Documents ON NETworks. Pronounced as if in French, this acronym sounds like *Fors données* (beyond data).
[4] Otherwise it is regarded as a plain text.

`<MODREF>`, `</MODREF>`
: These are tags to specify a module import. An imported module must be put between the tags.

`<CONSTRAINT>`, `</CONSTRAINT>`
: These are tags to specify a constraint. A `<CONSTRAINT>` tag must include information about the context of the constraint, the tool for solving the constraint, and so on.

`<TARGET>`, `</TARGET>`
: These are tags to describe an additional information required to solve a constraint. The URL of a part that describes an additional information must be put between the tags.

Like HTML, a URL is used to refer to a specification written in Forsdonnet. The protocol field of such a URL is "`fors`"[5]. For example,

```
fors://syphon.is.tsukuba.ac.jp/fermat/lemma3791
```

As Forsdonnet is an extension of HTML, Forsdonnet inherits various features from HTML. A document written in Forsdonnet is able to include not only codes in the language CafeOBJ, but also explanations, graphics or Java applets.

We show an example in Forsdonnet below.

```
<MODULE name=MY-RAT keyword=XYZ>
module MY-RAT {
    ... a definition of rational numbers ...
}
</MODULE>
```

Suppose the above module `MY-RAT` is described in a file named `my-rat.fdn` on a machine named `cafe.jaist.ac.jp`. A `<MODULE>` tag is used to specify the module. The name `MY-RAT` given as the attribute `name` of the tag `<MODULE>` is to be used when this module is imported elsewhere (see the document example below).

```
<HTML>
<HEAD><TITLE>FACT</TITLE></HEAD>
<BODY>
... explanations in natural language and diagrams ...
<MODULE name=FACT>
module FACT {
  pr (<MODREF context=fors://cafe.jaist.ac.jp/my-rat.fdn#MY-RAT>
      MY-RAT</MODREF>)
  op _! : Nat -> NzNat { memo }
```

[5]The need for such a protocol is rather technical. It acts in many respects exactly like the http protocol, but there are subtle differences.

```
  op fact : Nat -> NzNat
  eq 0 ! = 1 .
  eq N:NzNat ! = N * (p N !) .
  eq fact(0) = 1 .
  eq fact(N:NzNat) = N * fact(p N) .
}
</MODULE>
<TARGET name=CMD>
red 10 ! .
</TARGET>
<CONSTRAINT context=#FACT solver=CafeOBJ target=#CMD>
</CONSTRAINT>
</BODY>
</HTML>
```

The module FACT imports the module MY-RAT by giving the URL and the "name" MY-RAT as the attribute context of the tag <MODREF>[6]. The meaning of the attributes of <CONSTRAINT> is as follows.

context This attribute refers to the definitions (signatures, axioms, and so on) used to solve the constraint. Its value is usually a URL of a module: in the example above, it refers to the module FACT within the same document[7].

solver This attribute specifies an engine (tool) to solve the constraint. In the example, it is the system CafeOBJ.

target This attribute refers to a parameter to be passed to the solver. The reference is to a URL together with a part enclosed by the tag <TARGET>. In the example, it refers to the target CMD of the same document.

3.3. Solving Constraints

Solving a constraint is a task of a tool specified by the solver attribute of the <CONSTRAINT> tag. The result is inserted between the <CONSTRAINT> and </CONSTRAINT> tags. Since an inserted result can also be used as a target of another constraint, constraints can be nested and constraints are solved interactively.

The following is a simple example that demonstrates that natural numbers under addition defined in the module TINY-NAT satisfy the associative law. It involves solving two constraints. First the constraint C1 is solved to generate a structural induction scheme, whose solution gives a proof of the original goal. Then we solve the constraint C2, namely, the induction scheme just generated.

```
<MODULE name=TINY-NAT>
```

[6] By making use of name attributes, the user can distinguish two modules of the same name within a single file. During a development it is sometimes convenient to have several variants of a module in a file, and this feature is introduced for such occasions.

[7] To be more precise, the module below the <MODULE> tag whose name attribute is FACT.

```
module! TINY-NAT {
  [ Nat ]
  op 0 : -> Nat
  op s_ : Nat -> Nat {prec: 1}
  op _+_ : Nat Nat -> Nat
  vars M N : Nat
  eq M + 0 = M .
  eq M + s N = s(M + N) .
}
</MODULE>

<MODULE name=ASSOC>
module! ASSOC {
  <TARGET name=PR1>
  protecting (<MODREF context=#TINY-NAT>TINY-NAT</MODREF>)
  </TARGET>
  vars L M N : Nat
  eq L + (M + N) = (L + M) + N .
}
</MODULE>

<CONSTRAINT name=C1 context=#ASSOC target=#PR1 solver=obligation>
</CONSTRAINT>
<CONSTRAINT name=C2 context=#ASSOC target=#C1 solver=CafeOBJ>
</CONSTRAINT>
```

A "proof obligation generator"[5] extracts assertions associated with some constructs of the language CafeOBJ. In this example, it formulates in the syntax of the language CafeOBJ a sufficient condition to ensure that the import mode `protecting` is respected. Since the importing module `ASSOC` has a single new equation on the imported sort `Nat` and no new operator, for this import to be protecting, it is enough to show that the new equation is a theorem of the axioms of `NAT`[8].

In the constraint `C1`, the `solver` attribute is `obligation`[9], which refers to a proof obligation generator[10]. The `context` attribute says that the module context, in the sense of the system CafeOBJ, is the module `ASSOC`. The `target` attribute refers to the import declaration of `TINY-NAT` in that module. Read these attributes as an instruction, "extract the proof obligations of that import, and formulate them with structural induction". By solving `C1`, you get a text inserted below the `<CONSTRAINT>` tag, as[11]:

[8] For details of problem generators, see [5].

[9] These names and references may be changed by administration tools implemented as Netscape contents (see Section 4.2). These tools also determine what tools are installed on which machines.

[10] Strictly, `obligation` not only extracts assertions. In one step it generates induction schemes, as the result below shows.

[11] The actual names of the constants are a little more criptic. Also, note that the generated scheme uses induction on the last variable. This is by default, and may be changed by a parameter passed to the generator in the form of another attribute.

```
<CONSTRAINT name=C1 context=#ASSOC target=#PR1 solver=obligation>
open TINY-NAT
ops l m : -> Nat .
** base case
reduce l + (m + 0) == (l + m) + 0 .
** induction step
op n : -> Nat .
eq l + (m + n) = (l + m) + n .
reduce l + (m + s n) == (l + m) + s n .
close
</CONSTRAINT>
```

The new equation introduced by ASSOC, in fact, is exactly the associative law. Which implies that discharging this obligation is equivalent to proving associativity.

We turn to solving the constraint C2. The solver attribute is CafeOBJ, which refers to the system CafeOBJ. The context attribute is ASSOC, as before. the target attribute is C1, which was the name of the constraint just solved. Read these attributes as an instruction, "Invoke the system CafeOBJ, feed modules used in the context ASSOC, and then feed the texts (the command sequence) of the contraint C1"[12].

```
<CONSTRAINT name=C2 context=#ASSOC target=#C1 solver=CafeOBJ>
-- opening module TINY-NAT.. done._*
-- reduce in % : l + (m + 0) == (l + m) + 0
true : Bool
(0.000 sec for parse, 3 rewrites(0.000 sec), 11 match attempts)
-- reduce in % : l + (m + s n) == (l + m) + s n
true : Bool
(0.000 sec for parse, 5 rewrites(0.000 sec), 27 match attempts)
</CONSTRAINT>
```

By this result it is proven that Nat satisfies the associative law. In this manner, the proofs of properties are constructed interactively.

4. On the Implementation

We have finished implementing an integrated environment that is based on our framework. In this environment, users can write and explain specifications, solve constraints and see specifications on worldwide networks.

Here are a couple of basic principles behind the implementation.

- Standard tools, such as Emacs and Netscape, are used as they are. This decision lead to some drawbacks. For example, we could not incorporate Forsdonnet editors into Netscape: as a result, Netscape may be used only for browsing purposes. On

[12]Note here that our environment takes care of all the drudgery: it decides which modules are needed for a given context; it fetches those modules scattered over the network; it finds where the CafeOBJ system is running; it sends all the necessary data there; and it gets the output back to the user's machine.

the other hand, this decision made the implementation feasible and fairly rapid. It does not make much sense to implement a new browser from scratch, which, as network tools go, will become rapidly obsolete. Moreover, the decision made our environment robust. So long as upper compatibility is maintained, our environment can run with new versions of browsers and daemons without changing anything.

- Emacs and Netscape are two important interface tools and it is the user, not us, who decide to use which. Hence we must supply an environment for both. A question is, why only two? There may be other interfaces that will become as important, but at the time of this decision (and still at the time of writing this paper) we regard these two alternatives as adequate for all practical purposes.

  But it is not practical to give these two interfaces the identical functions. We regard Emacs as a developer's interface, and assume Netscape to be mainly a supervisor's interface. If you happen to be both, you need both of them, but it is unlikely for you, for example, to debug a programme code with Netscape. This classification seems to give a little justification for our decisions that (a) Netscape does not have Forsdonnet editors, as stated above, and that (b) Emacs is not equipped with system administration facilities.

- Behind the scene and whenever necessary, a Forsdonnet document are converted into an HTML document where special tags, such as `<MODULE>`, are interpreted as (sets of) standard HTML tags. This decision allows us to use Emacs and Netscape as they are, which is exactly what we want, in view of the above principle.

- One major consideration is how to cope with firewalls. We simply decided that we only use ways of communications that do not hit firewalls. On the negative side, this decision made interactions sometimes slower and/or restricted. On the positive side, our environment became suitable for industrial sites.

The environment consists of server sides and client sides. An http daemon must reside on both sides, for transmitting requests from client sides to server sides using common gateway interface (CGI). Note that in this setting the requests can go through general firewalls.

4.1. Server Sides

The server sides of the environment are composed of distributed objects in Java. All the objects belong to one of the categories listed below, and may reside on different machines.

Forsdonnet Object
: This is a persistent object that encapsulates a Forsdonnet file. It provides operations to save/load a Forsdonnet file into/from a remote file system, extract a specification from a Forsdonnet document, translate a Forsdonnet document into an HTML document and so on.

Session Object
: This is a transient object that acts as proxy for the system CafeOBJ or a prover,

and relays messages between a client and the tool. When a constraint is solved, this object invokes the tool, collects modules needed for the solving, then feeds the modules to the tool.

Repository Object
This is a persistent object that encapsulates meta-data for developing specifications. This object provides operations to search modules on several conditions, such as keywords, development status, and plain text matching.

4.2. Client Sides

The client sides of the environment are implemented on Netscape and Emacs. As the software can work on a standard Netscape or Emacs, the user need not prepare any special software.

Netscape Client
The software for Netscape consists essentially of contents (see Figure 9.2 for a sample window), and is written in HTML and JavaScript. It is loaded from a machine where an http daemon resides. The user is given menues, whose items are categorised as follows.

Forsdonnet: Menu items to create, edit[13], save and delete a specification in Forsdonnet. These operations are at the level of files, and are not concerned with what are in them. By convention a Forsdonnet file has a suffix ".fdn".

Browser: Menu items to browse a specification in Forsdonnet and to solve a constraint. When browsed, the tags particular to Forsdonnet are converted to HTML tags, and CafeOBJ codes are highlighted. A constraint is converted to a button, a push of which results in constraint solving[14].

Repository: Menu items to search modules. A keyword search scans the attribute `keyword` (if given) of each `<MODULE>` tag. A search with general text matching involves reading every line of every document.

Administration: Menu items to administrate the system. An example is an operation to define or change the tool configuration.

Emacs Client
The software for Emacs is written in Emacs Lisp and made out of w3.el, an HTML browser package. The features of this package is kept intact, so the user can also use this software as a plain HTML browser.

This software owns the same functions as the software for Netscape, except those in the administration category. As is usual with Emacs Lisp packages, the software defines key bindings for operations specific to Forsdonnet manipulation, such as constraint solving.

[13] You may only edit Forsdonnet files as plain texts. There is no special support.
[14] Note that these operations are for browsing, not for editing. When a constraint is solved, the result is displayed in a pop-up window, and is discarded when the browsing ends.

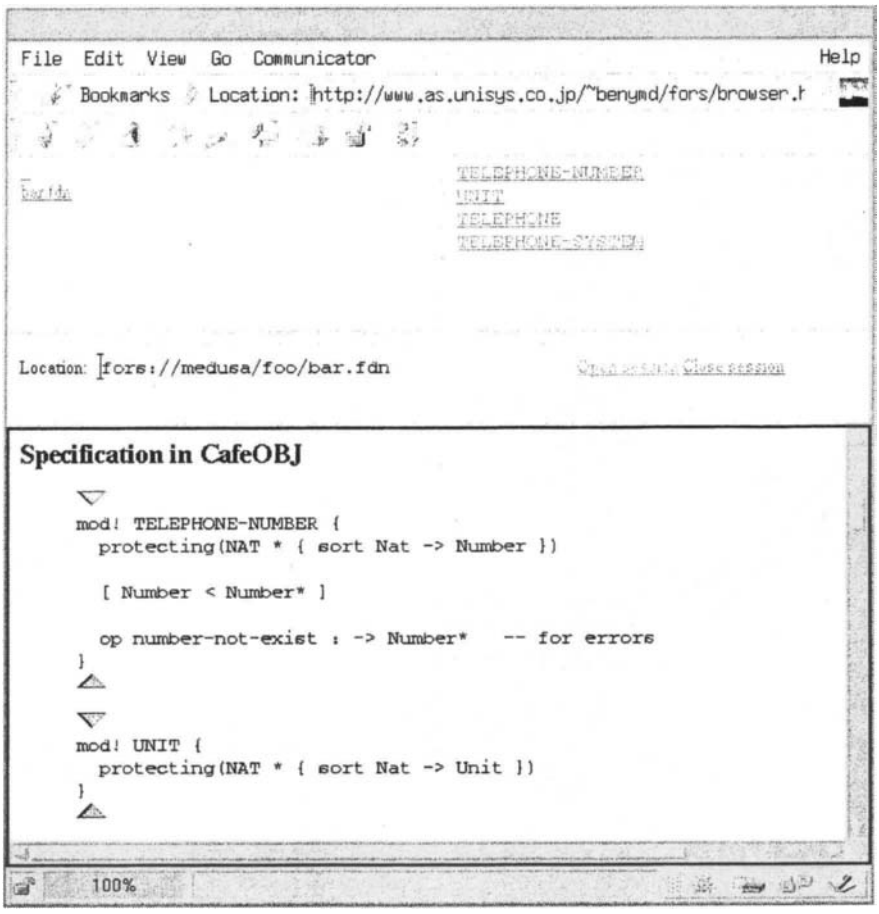


Figure 9.2. Netscape Client Window

Furthermore, this software has useful functions to edit tags of Forsdonnet. As an example, some of the tags are generated semi-automatically, using default attributes. It also has functions to hide or reveal tags systematically or locally, so that you can change from browsing task to editing task, and vice versa, effortlessly.

5. Conclusions

In Section 1, we brought up three problems — counting out the one discharged by assumption — concerning specification development in a distributed environment. The framework that we proposed gave the following solutions.

1. Modules distributed over a network can be identified explicitly and uniquely with the help of URLs.

2. A relation between modules that are put on different machines can be specified as a `<MODREF>` tag within one of the modules. In the case of views, target and source modules can be specified likewise.

3. The question of where to invoke the system CafeOBJ or a prover, and which one if there are several variants, can be answered as a `<CONSTRAINT>` tag in the relevant document, coupled with the tool configuration defined at the level of system administration[15].

For the user, the need to supply extra tags is an overhead, compared with the case when he is working alone within a single machine. With this proviso, thanks to the above solutions, the user can develop specifications in a distributed environment without the burden of scavenging through all those possible sites, of fearing name conflicts, of copying all those submodules, of visiting and starting provers, of copying and inspecting results on his own machine, and so on. Anyway, if you want to utilise distributed resources fully, an overhead for specifying locations and check/matching names is inevitable. Moreover, a couple of experiments have convinced us that the tagging consumes trivially little time when put in an entire development scene.

Furthermore, the framework has various flexibilities.

- Because Forsdonnet is defined as an extension to HTML, a document written in Forsdonnet can include not only specification (code)s but also explanations in natural languages, graphics, sounds, Java applets, links to another document, etc.

- Since constraints can be nested, users can solve constraints step by step rather than all at once, and in any order such as bottom up, top down, or their hybrid.

- Because of generality of the constraint-solving mechanism, it is easy to incorporate a new prover into the framework. It is even possible to support a language other than CafeOBJ in the framework, as long as the language has a similar module structure.

The integrated environment we implemented is based on the framework and is characterised as follows.

- The user can use standard tools such as Netscape and Emacs and work with a familiar interface. He also uses familiar URLs in locating things, as with most web tools.

- The environment works even if a general firewall is in the network. This situation is often found in practice, and is a cause of frustration. As long as a client machine connects with a network, even if the network is secure, a user of the machine can use the environment.

[15] It is also possible to customise the configuration at the level of users.

- The system provides operations to search modules on several conditions such as (predefined) keywords, development status, systematic text matching and so on. These search facilities are especially userful for development-in-the-large.

- Because useful functions to edit tags of Forsdonnet are provided, the productivity of specification developers is improved.

Among related works, the most interesting one is the Tatami system being developed at UCSD by Goguen et al.[3]. Their system supports, in particular, distributed proof construction on networks. They use a different scheme to organise proofs, and have realised flexibility without compromising the rigour of proofs. We believe it is of mutual benefit to compare two systems and to discuss pros and cons.

As a future work, we are considering visualising relations between constraints. In other words, we plan to represent proof trees visually. If such a visualisation tool is developed, the user will be able to get a clearer view of the whole verification process.

With regard to implementation, we are planning to apply XSL (eXtensible Stylesheet Language) to the presentation of Forsdonnet.

Acknowledgments

We would like to thank Masami Hagiya, who gave us the basic direction of this work. We also appreciate valuable comments from anonymous referees who patiently read an earlier draft.

REFERENCES

1. Futatsugi, K. and Diaconescu, R., *CafeOBJ Report*, World Scientific, 1998
2. Futatsugi, K. and Nakagawa, A.T., "An Overview of CAFE Specification Environment — an algebraic approach for creating, verifying and maintaining formal specifications over networks —", in *The Proceedings of the (First) IEEE International Conference on Formal Engineering Methods*, 1997
3. Goguen, J., Lin, K., Rosu, G., Mori, A., and Warinschi, B., "An Overview of the Tatami Project", in this volume, Elsevier, 2000.
4. Hagiya, M. and Kakuno, H., "Proving as Editing". in *The Proceedings of User Interfaces of Theorem Provers*, York, 1996
5. Ishiguro, M. and Nakagawa, A.T., "Proof Assistance for Equational Specifications Based on Proof Obligations", in this volume, Elsevier, 2000
6. Nakagawa, A.T., Sawada, T., and Futatsugi, K., *CafeOBJ Manual*, a technical report JAIST/SRA, 1997
7. Nakagawa, A.T. and Seo, A., "Directing Proofs by Documenting and Pointing: CafeOBJ on Networks". in *The Proceedings of User Interfaces of Theorem Provers*, Sophia Antipolis, 1997

This Page Intentionally Left Blank